Boost.Build V2 User Manual

Boost.Build V2 User Manual

Table of Contents

How to use this document 1
Installation 2
Tutorial 4
- Hello, world 4
- Properties 4
- Project Hierarchies 6
- Dependent Targets 7
- Testing 8
- Static and shared libaries 8
- Conditions and alternatives 9
- Prebuilt targets 10

Overview 11
- Concepts 11
- Boost.Jam Language 12
- Configuration 15
- Invocation 16
- Declaring Targets 19
- Projects 22
- The Build Process 24

Common tasks 26
- Programs 26
- Libraries 26
- Alias 28
- Installing 28
- Testing 30
- Custom commands 31
- Precompiled Headers 32
- Generated headers 32
- Cross-compilation 33

Reference 34
- General information 34
- Builtin rules 34
- Builtin features 35
- Builtin tools 38
- Build process 46
- Definitions 47

Extender Manual 52
- Introduction 52
- Example: 1-to-1 generator 54
- Target types 55
- Tools and generators 56
- Features 58
- Main target rules 60
- Toolset modules 61

Frequently Asked Questions 63
- How do I get the current value of feature in Jamfile? 63
- I am getting a "Duplicate name of actual target" error. What does that mean? 63
- Accessing environment variables 64
- How to control properties order? 64
- How to control the library linking order on Unix? 65
- Can I get capture external program output using a Boost.Jam variable? 65
- How to get the project root (a.k.a. Jamroot) location? 65
- How to change compilation flags for one file? 65
- Why are the `dll-path` and `hardcode-dll-paths` properties useful? 66
- Targets in site-config.jam 67
- Header-only libraries 67

A. Boost.Jam Documentation 68

Introduction 68
Building BJam 68
Using BJam 73
Language 76
Miscellaneous 94
History 97
B. Differences to Boost.Build V1 101
Configuration 101
Writing Jamfiles 101
Build process 101
Index 102

List of Tables

1. Search paths for configuration files 15
2. 18
3. 23
A.1. Supported Toolsets 69

How to use this document

If you've just found out about Boost.Build V2 and want to know if it will work for you, start with *Tutorial*. You can continue with *Overview*. When you're ready to try Boost.Build in practice, go to *Installation*.

If you are about to use Boost.Build on your project, or already using it and have a problem, look at *Overview*.

If you're trying to build a project which uses Boost.Build, see *Installation* and then read about the section called “Invocation”.

If you have questions, please post them to our mailing list (http://boost.org/more/mailing_lists.htm#jamboost). The mailing list is also mirrowed to newsgroup news://news.gmane.org/gmane.comp.lib.boost.build.

Installation

This section describes how to install Boost.Build from a released Boost source distribution or CVS image. [1] All paths are given relative to the *Boost.Build v2 root directory*, which is located in the `tools/build/v2` subdirectory of a full Boost distribution. [2]

1. Boost.Build uses Boost.Jam, an extension of the Perforce Jam portable **make** replacement. The recommended way to get Boost.Jam is to **download a prebuilt executable** from SourceForge. If a prebuilt executable is not provided for your platform or you are using Boost's sources in an unreleased state, it may be necessary to build **bjam** from sources included in the Boost source tree.

2. To install Boost.Jam, copy the executable, called **bjam** or **bjam.exe** to a location accessible in your `PATH`. Go to the Boost.Build root directory and run **bjam `--version`**. You should see:

```
- - - - - - - - - - - -Boost.Build -V2 -(Milestone -N)
- - - - - - - - - - - -Boost.Jam -xx.xx.xx -
- - - - - - - - - - -
```

   where N is the version of Boost.Build you're using.

3. Configure Boost.Build to recognize the build resources (such as compilers and libraries) you have installed on your system. Open the `user-config.jam` file in the Boost.Build root directory and follow the instructions there to describe your toolsets and libraries, and, if necessary, where they are located.

4. You should now be able to go to the `example/hello/` directory and run **bjam** there. A simple application will be built. You can also play with other projects in the `example/` directory.

If you are using Boost's CVS state, be sure to rebuild **bjam** even if you have a previous version. The CVS version of Boost.Build requires the CVS version of Boost.Jam.

When **bjam** is invoked, it always needs to be able to find the Boost.Build root directory, where the interpreted source code of Boost.Build is located. There are two ways to tell **bjam** about the root directory:

- Set the environment variable `BOOST_BUILD_PATH` to the absolute path of the Boost.Build root directory.
- At the root directory of your project or in any of its parent directories, create a file called `boost-build.jam`, with a single line:

```
boost-build -/path/to/boost.build -;
```

Information for distributors

If you're planning to package Boost.Build for a Linux distribution, please follow these guidelines:

[1] Note that packages prepared for Unix/Linux systems usually make their own choices about where to put things and even which parts of Boost to include. When we say "released source distribution" we mean a distribution of Boost as released on its SourceForge project page.

[2] The Boost.Build subset of boost is also distributed separately, for those who are only interested in getting a build tool. The top-level directory of a Boost.Build distribution contains all the subdirectories of the `tools/build/v2` subdirectory from a full Boost distribution, so it is itself a valid Boost.Build root directory. It also contains the `tools/jam/src` subdirectory of a full Boost distribution, so you can rebuild Boost.Jam from source.

- Create a separate package for Boost.Jam.
- Create another package for Boost.Build, and make this package install all Boost.Build files to `/usr/share/boost-build` directory. After install, that directory should contain everything you see in Boost.Build release package, except for `jam_src` directory. If you're using Boost CVS to obtain Boost.Build, as opposed to release package, take everything from the `tools/build/v2` directory. For a check, make sure that `/usr/share/boost-build/boost-build.jam` is installed.

  Placing Boost.Build into `/usr/share/boost-build` will make sure that **bjam** will find Boost.Build without any additional setup.
- Provide a `/etc/site-config.jam` configuration file that will contain:

```
using -gcc -;
```

  You might want to add dependency from Boost.Build package to gcc, to make sure that users can always build Boost.Build examples.

If those guidelines are met, users will be able to invoke **bjam** without any explicit configuration.

Tutorial

This section will guide you though the most basic features of Boost.Build V2. We will start with the “Hello, world” example, learn how to use libraries, and finish with testing and installing features.

Hello, world

The simplest project that Boost.Build can construct is stored in `example/hello/` directory. The project is described by a file called `Jamroot` that contains:

```
exe -hello -: -hello.cpp -;
```

Even with this simple setup, you can do some interesting things. First of all, just invoking **bjam** will build the `hello` executable by compiling and linking `hello.cpp` . By default, debug variant is built. Now, to build the release variant of `hello`, invoke

```
bjam -release
```

Note that debug and release variants are created in different directories, so you can switch between variants or even build multiple variants at once, without any unnecessary recompilation. Let us extend the example by adding another line to our project's `Jamroot`:

```
exe -hello2 -: -hello.cpp -;
```

Now let us build both the debug and release variants of our project again:

```
bjam -debug -release
```

Note that two variants of `hello2` are linked. Since we have already built both variants of `hello`, hello.cpp will not be recompiled; instead the existing object files will just be linked into the corresponding variants of `hello2`. Now let us remove all the built products:

```
bjam ---clean -debug -release
```

It is also possible to build or clean specific targets. The following two commands, respectively, build or clean only the debug version of `hello2`.

```
bjam -hello2
bjam ---clean -hello2
```

Properties

To portably represent aspects of target configuration such as debug and release variants, or single- and multi-threaded builds, Boost.Build uses *features* with associated *values*. For example, the `debug-symbols` feature can have a value of `on` or `off`. A *property* is just a (feature, value) pair. When a user initiates a build, Boost.Build automatically translates the requested properties into appropriate command-line flags for invoking toolset components like compilers and linkers.

There are many built-in features that can be combined to produce arbitrary build configurations. The following command builds the project's `release` variant with inlining disabled and debug symbols enabled:

```
bjam -release -inlining=off -debug-symbols=on
```

Properties on the command-line are specified with the syntax:

```
feature-name=feature-value
```

The `release` and `debug` that we have seen in **bjam** invocations are just a shorthand way to specify values of the `variant` feature. For example, the command above could also have been written this way:

```
bjam -variant=release -inlining=off -debug-symbols=on
 - - - - - -
```

`variant` is so commonly-used that it has been given special status as an *implicit* feature— Boost.Build will deduce the its identity just from the name of one of its values.

A complete description of features can be found in the section called “Features and properties”.

Build Requests and Target Requirements

The set of properties specified on the command line constitute a *build request*—a description of the desired properties for building the requested targets (or, if no targets were explicitly requested, the project in the current directory). The *actual* properties used for building targets are typically a combination of the build request and properties derived from the project's `Jamroot` (and its other Jamfiles, as described in the section called “Project Hierarchies”). For example, the locations of `#included` header files are normally not specified on the command-line, but described in Jamfiles as *target requirements* and automatically combined with the build request for those targets. Multithread-enabled compilation is another example of a typical target requirement. The Jamfile fragment below illustrates how these requirements might be specified.

```
exe -hello
 - - - -: -hello.cpp
 - - - -: -<include>boost -<threading>multi
 - - - -;
```

When `hello` is built, the two requirements specified above will always be present. If the build request given on the **bjam** command-line explictly contradicts a target's requirements, the target requirements usually override (or, in the case of “free”” features like `<include>`, [1] augments) the build request.

> **Tip**
>
> The value of the `<include>` feature is relative to the location of `Jamroot` where it is used.

Project Attributes

If we want the same requirements for our other target, `hello2` , we could simply duplicate them. However, as projects grow, that approach leads to a great deal of repeated boilerplate in Jamfiles. Fortunately, there's a better way. Each project can specify a set of *attributes*, including requirements:

```
project
 - - - -: -requirements -<include>/home/ghost/Work/boost -<threading>multi
```

[1] See the section called “Feature Attributes”

```
 - - - -;

exe -hello -: -hello.cpp -;
exe -hello2 -: -hello.cpp -;
```

The effect would be as if we specified the same requirement for both `hello` and `hello2`.

Project Hierarchies

So far we have only considered examples with one project —a. with one user-written Boost.Jam file, `Jamroot`). A typical large codebase would be composed of many projects organized into a tree. The top of the tree is called the *project root*. Every subproject is defined by a file called `Jamfile` in a descendant directory of the project root. The parent project of a subproject is defined by the nearest `Jamfile` or `Jamroot` file in an ancestor directory. For example, in the following directory layout:

```
top/
 - -|
 - -+-- -Jamroot
 - -|
 - -+-- -app/
 - -| - - - -|
 - -| - - - -+-- -Jamfile
 - -| - - - -`-- -app.cpp
 - -|
 - -`-- -util/
 - - - - - - -|
 - - - - - - -+-- -foo/
 - - - - - - -. - - - -|
 - - - - - - -. - - - -+-- -Jamfile
 - - - - - - -. - - - -`-- -bar.cpp
```

the project root is `top/`. The projects in `top/app/` and `top/util/foo/` are immediate children of the root project.

Note

When we refer to a “Jamfile,” set in normal type, we mean a file called either `Jamfile` or `Jamroot`. When we need to be more specific, the filename will be set as “`Jamfile`” or “`Jamroot`.”

Projects inherit all attributes (such as requirements) from their parents. Inherited requirements are combined with any requirements specified by the subproject. For example, if `top/Jamroot` has

```
<include>/home/ghost/local
```

in its requirements, then all of its subprojects will have it in their requirements, too. Of course, any project can add include paths to those specified by its parents. [2] More details can be found in the section called “Projects”.

Invoking **bjam** without explicitly specifying any targets on the command line builds the project rooted in the current directory. Building a project does not automatically cause its subprojects to be built unless the parent project's Jamfile explicitly requests it. In our example, `top/Jamroot` might contain:

```
build-project -app -;
```

which would cause the project in `top/app/` to be built whenever the project in `top/` is built. However, targets in `top/util/foo/` will be built only if they are needed by targets in `top/` or `top/app/`.

[2] Many features will be overridden, rather than added-to, in subprojects. See the section called “Feature Attributes” for more information

Dependent Targets

When a building a target `X` depends on first building another target `Y` (such as a library that must be linked with *X*), `Y` is called a *dependency* of `X` and `X` is termed a *dependent* of `Y`.

To get a feeling of target dependencies, let's continue the above example and see how `top/app/Jamfile` can use libraries from `top/util/foo`. If `top/util/foo/Jamfile` contains

```
lib -bar -: -bar.cpp -;
```

then to use this library in `top/app/Jamfile`, we can write:

```
exe -app -: -app.cpp -../util/foo//bar -;
```

While `app.cpp` refers to a regular source file, `../util/foo//bar` is a reference to another target: a library `bar` declared in the Jamfile at `../util/foo`.

> **Tip**
>
> Some other build system have special syntax for listing dependent libraries, for example `LIBS` variable. In Boost.Build, you just add the library to the list of sources.

Suppose we build `app` with:

```
bjam -app -optimization=full -define=USE_ASM
 - - - -
```

Which properties will be used to build `foo`? The answer is that some features are *propagated*—Boost.Build attempts to use dependencies with the same value of propagated features. The `<optimization>` feature is propagated, so both `app` and `foo` will be compiled with full optimization. But `<define>` is not propagated: its value will be added as-is to the compiler flags for `a.cpp`, but won't affect `foo`.

Let's improve this project further. The library probably has some headers that must be used when compiling `app.cpp`. We could manually add the necessary `#include` paths to `app` 's requirements as values of the `<include>` feature, but then this work will be repeated for all programs that use `foo`. A better solution is to modify `util/foo/Jamfile` in this way:

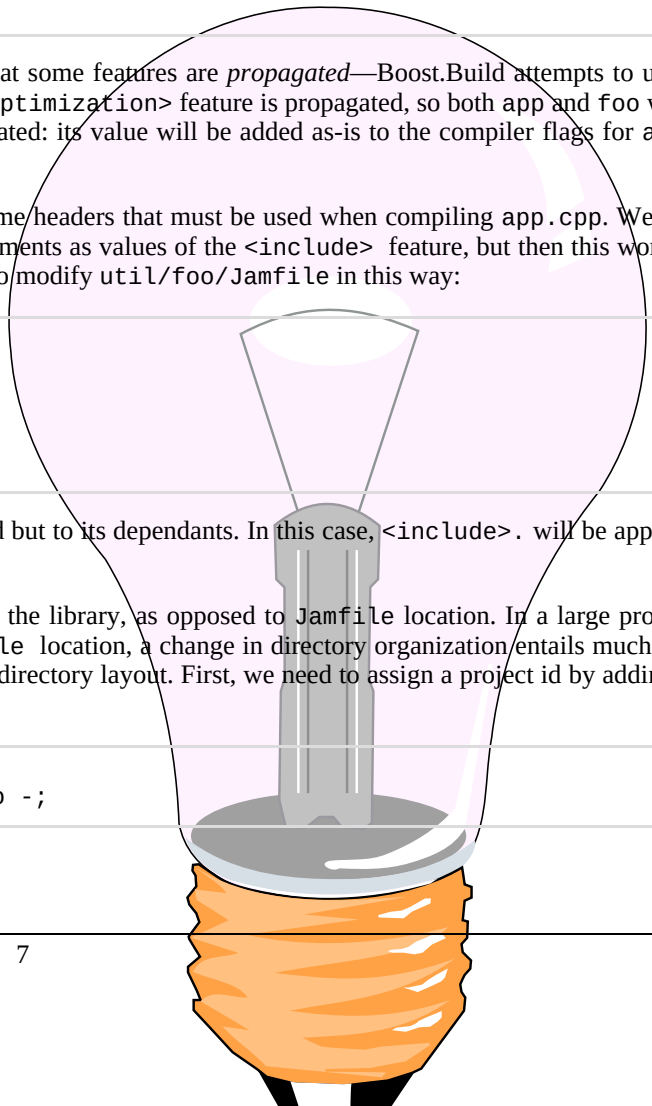
```
project
 - - - -: -usage-requirements -<include>.
 - - - -;

lib -foo -: -foo.cpp -;
```

Usage requirements are applied not to the target being declared but to its dependants. In this case, `<include>.` will be applied to all targets that directly depend on `foo`.

Another improvement is using symbolic identifiers to refer to the library, as opposed to `Jamfile` location. In a large project, a library can be used by many targets, and if they all use `Jamfile` location, a change in directory organization entails much work. The solution is to use project ids—symbolic names not tied to directory layout. First, we need to assign a project id by adding this code to `Jamroot`:

```
use-project -/library-example/foo -: -util/foo -;
```

Second, we modify `app/Jamfile` to use the project id:



```
exe -app -: -app.cpp -/library-example/foo//bar -;
```

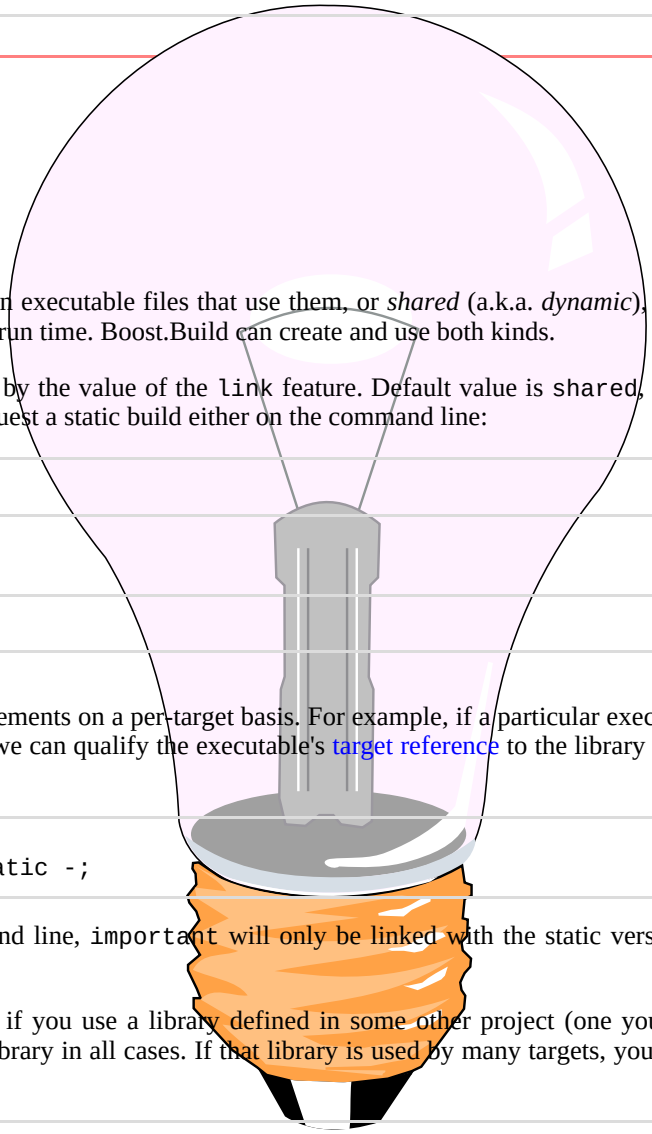
The `/library-example/foo//bar` syntax is used to refer to the target `bar` in the project with id `/library-example/foo`. We've achieved our goal—if the library is moved to a different directory, only `Jamroot` must be modified. Note that project ids are global—two Jamfiles are not allowed to assign the same project id to different directories.

> **Tip**
>
> If you want all applications in some project to link to a certain library, you can avoid having to specify it directly the sources of every target by using the `<library>` property. For example, if `/boost/filesystem//fs` should be linked to all applications in your project, you can add `<library>/boost/filesystem//fs` to the project's requirements, like this:

```
project
 - - -: -requirements -<library>/boost/filesystem//fs
 - - -;
```

Testing

Static and shared libaries



Libraries can be either *static*, which means they are included in executable files that use them, or *shared* (a.k.a. *dynamic*), which are only referred to from executables, and must be available at run time. Boost.Build can create and use both kinds.

The kind of library produced from a `lib` target is determined by the value of the `link` feature. Default value is `shared`, and to build a static library, the value should be `static`. You can request a static build either on the command line:

```
bjam -link=static
```

or in the library's requirements:

```
lib -l -: -l.cpp -: -<link>static -;
```

We can also use the `<link>` property to express linking requirements on a per-target basis. For example, if a particular executable can be correctly built only with the static version of a library, we can qualify the executable's target reference to the library as follows:

```
exe -important -: -main.cpp -helpers/<link>static -;
```

No matter what arguments are specified on the **bjam** command line, `important` will only be linked with the static version of `helpers`.

Specifying properties in target references is especially useful if you use a library defined in some other project (one you can't change) but you still want static (or dynamic) linking to that library in all cases. If that library is used by many targets, you *could* use target references everywhere:

```
exe -e1 -: -e1.cpp -/other_project//bar/<link>static -;
exe -e10 -: -e10.cpp -/other_project//bar/<link>static -;
```

but that's far from being convenient. A better approach is to introduce a level of indirection. Create a local alias target that refers to the static (or dynamic) version of `foo`:

```
alias -foo -: -/other_project//bar/<link>static -;
exe -e1 -: -e1.cpp -foo -;
exe -e10 -: -e10.cpp -foo -;
```

The `alias` rule is specifically used to rename a reference to a target and possibly change the properties.

Tip

When one library uses another, you put the second library in the source list of the first. For example:

```
lib -utils -: -utils.cpp -/boost/filesystem//fs -;
lib -core -: -core.cpp -utils -;
exe -app -: -app.cpp -core -;
```

This works no matter what kind of linking is used. When `core` is built as a shared library, it is linked directly into `utils`. Static libraries can't link to other libraries, so when `core` is built as a static library, its dependency on `utils` is passed along to `core`'s dependents, causing `app` to be linked with both `core` and `utils` .

Note

(Note for non-UNIX system). Typically, shared libraries must be installed to a directory in the dynamic linker's search path. Otherwise, applications that use shared libraries can't be started. On Windows, the dynamic linker's search path is given by the `PATH` environment variable. This restriction is lifted when you use Boost.Build testing facilities—the `PATH` variable will be automatically adjusted before running the executable.

Conditions and alternatives

Sometimes, particular relationships need to be maintained among a target's build properties. For example, you might want to set specific `#define` when a library is built as shared, or when a target's `release` variant is built. This can be achieved using *conditional requirements*.

```
lib -network -: -network.cpp
 - - - -: -<link>shared:<define>NEWORK_LIB_SHARED
 - - - - -<variant>release:<define>EXTRA_FAST
 - - - -;
```

In the example above, whenever `network` is built with `<link>shared`, `<define>NEWORK_LIB_SHARED` will be in its properties, too. Also, whenever its release variant is built, `<define>EXTRA_FAST` will appear in its properties.

Sometimes the ways a target is built are so different that describing them using conditional requirements would be hard. For example, imagine that a library actually uses different source files depending on the toolset used to build it. We can express this situation using *target alternatives*:

```
lib -demangler -: -dummy_demangler.cpp -; - - - - - - - - - - - - - - - - - - - - -# -altern
ative -1
lib -demangler -: -demangler_gcc.cpp -: -<toolset>gcc -; - - -# -alternative -2
lib -demangler -: -demangler_msvc.cpp -: -<toolset>msvc -; -# -alternative -3
```

When building `demangler`, Boost.Build will compare requirements for each alternative with build properties to find the best match. For example, when building with `<toolset>gcc` alternative 2, will be selected, and when building with `<toolset>msvc`

alternative 3 will be selected. In all other cases, the most generic alternative 1 will be built.

Prebuilt targets

To link to libraries whose build instructions aren't given in a Jamfile, you need to create `lib` targets with an appropriate `file` property. Target alternatives can be used to associate multiple library files with a single conceptual target. For example:

```
# -util/lib2/Jamfile
lib -lib2
 - - - -:
 - - - -: -<file>lib2_release.a -<variant>release
 - - - -;

lib -lib2
 - - - -:
 - - - -: -<file>lib2_debug.a -<variant>debug
 - - - -;
```

This example defines two alternatives for `lib2`, and for each one names a prebuilt file. Naturally, there are no sources. Instead, the `<file>` feature is used to specify the file name.

Once a prebuilt target has been declared, it can be used just like any other target:

```
exe -app -: -app.cpp -../util/lib2//lib2 -;
```

As with any target, the alternative selected depends on the properties propagated from `lib2`'s dependants. If we build the release and debug versions of `app` will be linked with `lib2_release.a` and `lib2_debug.a` , respectively.

System libraries—those that are automatically found by the toolset by searching through some set of predetermined paths—should be declared almost like regular ones:

```
lib -pythonlib -: -: -<name>python22 -;
```

We again don't specify any sources, but give a `name` that should be passed to the compiler. If the gcc toolset were used to link an executable target to `pythonlib`, `-lpython22` would appear in the command line (other compilers may use different options).

We can also specify where the toolset should look for the library:

```
lib -pythonlib -: -: -<name>python22 -<search>/opt/lib -;
```

And, of course, target alternatives can be used in the usual way:

```
lib -pythonlib -: -: -<name>python22 -<variant>release -;
lib -pythonlib -: -: -<name>python22_d -<variant>debug -;
```

A more advanced use of prebuilt targets is described in the section called “Targets in site-config.jam”.

Overview

This section will provide the information necessary to create your own projects using Boost.Build. The information provided here is relatively high-level, and *Reference* as well as the on-line help system must be used to obtain low-level documentation (see `--help`).

Boost.Build actually consists of two parts - Boost.Jam, a build engine with its own interpreted language, and Boost.Build itself, implemented in Boost.Jam's language. The chain of events when you type **bjam** on the command line is as follows:

1. Boost.Jam tries to find Boost.Build and loads the top-level module. The exact process is described in the section called "Initialization"
2. The top-level module loads user-defined configuration files, `user-config.jam` and `site-config.jam` , which define available toolsets.
3. The Jamfile in the current directory is read. That in turn might cause reading of further Jamfiles. As a result, a tree of projects is created, with targets inside projects.
4. Finally, using the build request specified on the command line, Boost.Build decides which targets should be built and how. That information is passed back to Boost.Jam, which takes care of actually running the scheduled build action commands.

So, to be able to successfully use Boost.Build, you need to know only four things:

- How to configure Boost.Build
- How to declare targets in Jamfiles
- How the build process works
- Some Basics about the Boost.Jam language. See the section called "Boost.Jam Language".

Concepts

Boost.Build has a few unique concepts that are introduced in this section. The best way to explain the concepts is by comparison with more classical build tools.

When using any flavour of make, you directly specify *targets* and commands that are used to create them from other target. The below example creates `a.o` from `a.c` using a hardcoded compiler invocation command.

```
a.o: -a.c
 - - - -g++ --o -a.o --g -a.c
```

This is rather low-level description mechanism and it's hard to adjust commands, options, and sets of created targets depending on the used compiler and operating system.

To improve portability, most modern build system provide a set of higher-level functions that can be used in build description files. Consider this example:

```
add_program -("a", -"a.c")
```

This is a function call that creates targets necessary to create executable file from source file `a.c`. Depending on configured properties, different commands line may be used. However, `add_program` is higher-level, but rather thin level. All targets are created immediately when build description is parsed, which makes it impossible to perform multi-variant builds. Often, change in any build property requires complete reconfiguration of the build tree.

In order to support true multivariant builds, Boost.Build introduces the concept of *metatarget*—object that is created when build description is parsed and can be later called with specific build properties to generate actual targets.

Consider an example:

```
exe -a -: -a.cpp -;
```

When this declaration is parsed, Boost.Build creates a metatarget, but does not yet decides what files must be created, or what commands must be used. After all build files are parsed, Boost.Build considers properties requested on the command line. Supposed you have invoked Boost.Build with:

```
bjam -toolset=gcc -toolset=msvc
```

In that case, the metatarget will be called twice, once with `toolset=gcc` and once with `toolset=msvc`. Both invocations will produce concrete targets, that will have different extensions and use different command lines.

Another key concept is *build property*. Build property is a variable that affects the build process. It can be specified on the command line, and is passed when calling a metatarget. While all build tools have a similar mechanism, Boost.Build differs by requiring that all build properties are declared in advance, and providing a large set of properties with portable semantics.

The final concept is *property propagation*. Boost.Build does not require that every metatarget is called with the same properties. Instead, the "top-level" metatargets are called with the properties specified on the command line. Each metatarget can elect to augment or override some properties (in particular, using the requirements mechanism, see the section called “Requirements”). Then, the dependency metatargets are called with modified properties and produce concrete targets that are then used in build process. Of course, dependency metatargets maybe in turn modify build properties and have dependencies of their own.

For more in-depth treatment of the requirements and concepts, you may refer to SYRCoSE 2009 Boost.Build article.

Boost.Jam Language

This section will describe the basics of the Boost.Jam language— just enough for writing Jamfiles. For more information, please see the Boost.Jam documentation.

Boost.Jam has an interpreted, procedural language. On the lowest level, a Boost.Jam program consists of variables and *rules* (Jam term for function). They are grouped into modules—there is one global module and a number of named modules. Besides that, a Boost.Jam program contains classes and class instances.

Syntantically, a Boost.Jam program consists of two kind of elements—keywords (which have a special meaning to Boost.Jam) and literals. Consider this code:

```
a -= -b -;
```

which assigns the value `b` to the variable `a` . Here, `=` and `;` are keywords, while `a` and `b` are literals.

Warning

All syntax elements, even keywords, must be separated by spaces. For example, omitting the space character before `;` will lead to a syntax error.

If you want to use a literal value that is the same as some keyword, the value can be quoted:

```
a -= -"=" -;
```

All variables in Boost.Jam have the same type—list of strings. To define a variable one assigns a value to it, like in the previous example. An undefined variable is the same as a variable with an empty value. Variables can be accessed using the `$(variable)` syntax. For example:

```
a -= -$(b) -$(c) -;
```

Rules are defined by specifying the rule name, the parameter names, and the allowed value list size for each parameter.

```
rule -example
 -(
 - - - - -parameter1 -:
 - - - - -parameter2 -? -:
 - - - - -parameter3 -+ -:
 - - - - -parameter4 -*
 -)
 -{
 - - - -# -rule -body
 -}
 -
```

When this rule is called, the list passed as the first argument must have exactly one value. The list passed as the second argument can either have one value of be empty. The two remaining arguments can be arbitrarily long, but the third argument may not be empty.

The overview of Boost.Jam language statements is given below:

```
helper -1 -: -2 -: -3 -;
x -= -[ -helper -1 -: -2 -: -3 -] -;
```

This code calls the named rule with the specified arguments. When the result of the call must be used inside some expression, you need to add brackets around the call, like shown on the second line.

```
if -cond -{ -statements -} -[ -else -{ -statements -} -]
```

This is a regular if-statement. The condition is composed of:

- Literals (true if at least one string is not empty)
- Comparisons: `a` *`operator`* `b` where *`operator`* is one of `=`, `!=`, `<`, `>`, `<=` or `>=`. The comparison is done pairwise between each string in the left and the right arguments.
- Logical operations: `! a`, `a && b`, `a || b`
- Grouping: `( cond )`

```
for -var -in -list -{ -statements -}
```

Executes statements for each element in list, setting the variable `var` to the element value.

```
while -cond -{ -statements -}
```

Repeatedly execute statements while cond remains true upon entry.

```
return -values -;
```

This statement should be used only inside a rule and assigns `values` to the return value of the rule.

Warning

The `return` statement does not exit the rule. For example:

```
rule -test -( -)
{
 - - -if -1 -= -1
 - - -{
 - - - - - -return -"reasonable" -;
 - - -}
 - - -return -"strange" -;
}
```

will return `strange`, not `reasonable`.

```
import -module -;
import -module -: -rule -;
```

The first form imports the specified bjam module. All rules from that module are made available using the qualified name: *`module.rule`*. The second form imports the specified rules only, and they can be called using unqualified names.

Sometimes, you'd need to specify the actual command lines to be used when creating targets. In jam language, you use named actions to do this. For example:

```
actions -create-file-from-another
{
 - - - -create-file-from-another -$(<) -$(>)
}
```

This specifies a named action called `create-file-from-another`. The text inside braces is the command to invoke. The `$(<)` variable will be expanded to a list of generated files, and the `$(>)` variable will be expanded to a list of source files.

To flexibly adjust the command line, you can define a rule with the same name as the action and taking three parameters -- targets, sources and properties. For example:

```
rule -create-file-from-another -( -targets -* -: -sources -* -: -properties -* -)
{
 - - -if -<variant>debug -in -$(properties)
 - - -{
 - - - - - - -OPTIONS -on -$(targets) -= ---debug -;
 - - -}
}
actions -create-file-from-another
{
 - - - -create-file-from-another -$(OPTIONS) -$(<) -$(>)
}
```

In this example, the rule checks if certain build property is specified. If so, it sets variable `OPIONS` that is then used inside the action. Note that the variables set "on a target" will be visible only inside actions building that target, not globally. Were they set globally, using variable named `OPTIONS` in two unrelated actions would be impossible.

More details can be found in Jam reference, the section called “ Rules”.

Configuration

On startup, Boost.Build searches and reads two configuration files: `site-config.jam` and `user-config.jam`. The first one is usually installed and maintained by system administrator, and the second is for user to modify. You can edit the one in the top-level directory of Boost.Build installation or create a copy in your home directory and edit the copy. The following table explains where both files are searched.

Table 1. Search paths for configuration files

	site-config.jam	**user-config.jam**

Linux	`/etc`
	`$HOME`
	`$BOOST_BUILD_PATH`
		`$HOME`
		`$BOOST_BUILD_PATH`

Windows	`%SystemRoot%`
	`%HOMEDRIVE%%HOMEPATH%`
	`%HOME%`
	`%BOOST_BUILD_PATH%`
		`%HOMEDRIVE%%HOMEPATH%`
		`%HOME%`
		`%BOOST_BUILD_PATH%`

> **Tip**
>
> You can use the **--debug-configuration** option to find which configuration files are actually loaded.

Usually, `user-config.jam` just defines available compilers and other tools (see the section called “Targets in site-config.jam” for more advanced usage). A tool is configured using the following syntax:

```
using -tool-name -: -... -;
```

The `using` rule is given a name of tool, and will make that tool available to Boost.Build. For example,

```
using -gcc -;
```

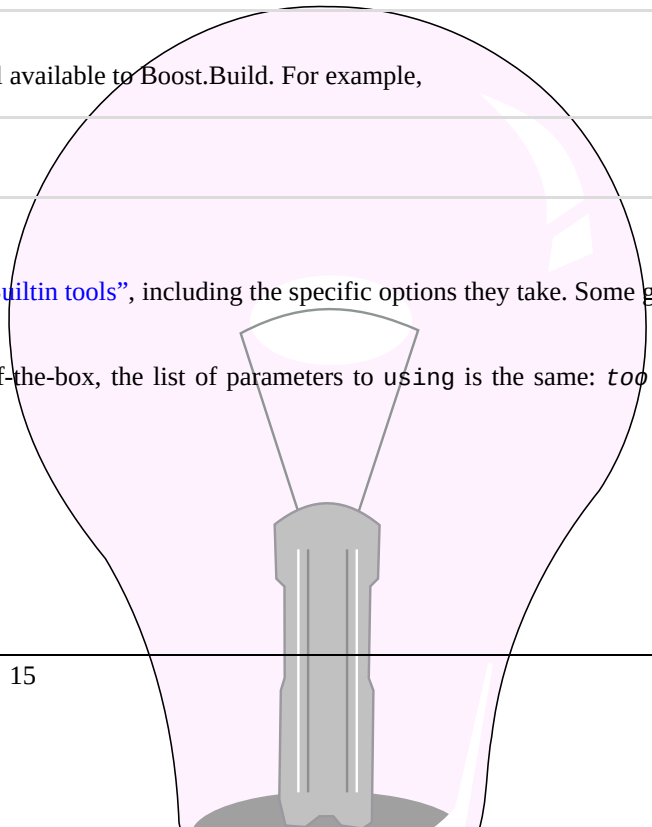
will make the GCC compiler available.

All the supported tools are documented in the section called “Builtin tools”, including the specific options they take. Some general notes that apply to most C++ compilers are below.

For all the C++ compiler toolsets Boost.Build supports out-of-the-box, the list of parameters to `using` is the same: *`toolset-name`*, *`version`*, *`invocation-command`*, and *`options`*.

If you have a single compiler, and the compiler executable

- has its “usual name” and is in the `PATH`, or
- was installed in a standard “installation directory”, or



- can be found using a global system like the Windows registry.

it can be configured by simply:

```
using -tool-name -;
```

If the compiler is installed in a custom directory, you should provide the command that invokes the compiler, for example:

```
using -gcc -: -: -g++-3.2 -;
using -msvc -: -: -"Z:/Programs/Microsoft -Visual -Studio/vc98/bin/cl" -;
```

Some Boost.Build toolsets will use that path to take additional actions required before invoking the compiler, such as calling vendor-supplied scripts to set up its required environment variables. When compiler executables for C and C++ are different, path to the C++ compiler executable must be specified. The command can be any command allowed by the operating system. For example:

```
using -msvc -: -: -echo -Compiling -&& -foo/bar/baz/cl -;
```

will work.

To configure several versions of a toolset, simply invoke the `using` rule multiple times:

```
using -gcc -: -3.3 -;
using -gcc -: -3.4 -: -g++-3.4 -;
using -gcc -: -3.2 -: -g++-3.2 -;
```

Note that in the first call to `using`, the compiler found in the `PATH` will be used, and there is no need to explicitly specify the command.

Many of toolsets have an *`options`* parameter to fine-tune the configuration. All of Boost.Build's standard compiler toolsets accept four options `cflags`, `cxxflags`, `compileflags` and `linkflags` as *`options`* specifying flags that will be always passed to the corresponding tools. Values of the `cflags` feature are passed directly to the C compiler, values of the `cxxflags` feature are passed directly to the C++ compiler, and values of the `compileflags` feature are passed to both. For example, to configure a **gcc** toolset so that it always generates 64-bit code you could write:

```
- - - - - - - -using -gcc -: -3.4 -: -: -<compileflags>-m64 -<linkflags>-m64 -;
```

Warning

Although the syntax used to specify toolset options is very similar to syntax used to specify requirements in Jamfiles, the toolset options are not the same as features. Don't try to specify a feature value in toolset initialization.

Invocation

To invoke Boost.Build, type **bjam** on the command line. Three kinds of command-line tokens are accepted, in any order:

options — Options start with either dash, or two dashes. The standard options are listed below, and each project may add additional options

properties — Properties specify details of what you want to build (e.g. debug or release variant). Syntactically, all command line tokens with equal sign in them are considered to specify properties. In the simplest form, property looks like ***feature*=*value***

target — All tokens that are neither options nor properties specify what targets to build. The available targets entirely depend on the project you are building.

Examples

To build all targets defined in Jamfile in the current directory with default properties, run:

```
bjam
```

To build specific targets, specify them on the command line:

```
bjam -lib1 -subproject//lib2
```

To request a certain value for some property, add *property=value* to the command line:

```
bjam -toolset=gcc -variant=debug -optimization=space
```

Options

Boost.Build recognizes the following command line options.

`--help` — Invokes the online help system. This prints general information on how to use the help system with additional --help* options.

`--clean` — Cleans all targets in the current directory and in any subprojects. Note that unlike the `clean` target in make, you can use `--clean` together with target names to clean specific targets.

`--clean-all` — Cleans all targets, no matter where they are defined. In particular, it will clean targets in parent Jamfiles, and targets defined under other project roots.

`--build-dir` — Changes build directories for all project roots being built. When this option is specified, all Jamroot files should declare project name. The build directory for the project root will be computed by concatanating the value of the `--build-dir` option, the project name specified in Jamroot, and the build dir specified in Jamroot (or `bin`, if none is specified).

The option is primarily useful when building from read-only media, when you can't modify Jamroot.

`--version` — Prints information on Boost.Build and Boost.Jam versions.

`-a` — Causes all files to be rebuilt.

`-n` — Do no execute the commands, only print them.

`-d+2` — Show commands as they are executed.

`-d0` — Supress all informational messages.

`-q` — Stop at first error, as opposed to continuing to build targets that don't depend on the failed ones.

`-j N`	Run up to *`N`* commands in parallel.
`--debug-configuration`	Produces debug information about loading of Boost.Build and toolset files.
`--debug-building`	Prints what targets are being built and with what properties.
`--debug-generators`	Produces debug output from generator search process. Useful for debugging custom generators.
`--ignore-config`	Do not load `site-config.jam` and `user-config.jam` configuration files.

Properties

In the simplest case, the build is performed with a single set of properties, that you specify on the command line with elements in the form ***`feature=value`***. The complete list of features can be found in the section called “Builtin features”. The most common features are summarized below.

Table 2.

Feature	Allowed values	Notes
variant	debug,release	
link	shared,static	Determines if Boost.Build creates shared or static libraries
threading	single,multi	Cause the produced binaries to be thread-safe. This requires proper support in the source code itself.
address-model	32,64	Explicitly request either 32-bit or 64-bit code generation. This typically requires that your compiler is appropriately configured. Please refer to the section called “C++ Compilers” and your compiler documentation in case of problems.
toolset	(Depends on configuration)	The C++ compiler to use. See the section called “C++ Compilers” for a detailed list.
include	(Arbitrary string)	Additional include paths for C and C++ compilers.
define	(Arbitrary string)	Additional macro definitions for C and C++ compilers. The string should be either `SYMBOL` or `SYMBOL=VALUE`
cxxflags	(Arbitrary string)	Custom options to pass to the C++ compiler.
cflags	(Arbitrary string)	Custom options to pass to the C compiler.
linkflags	(Arbitrary string)	Custom options to pass to the C++ linker.
runtime-link	shared,static	Determines if shared or static version of C and C++ runtimes should be used.

If you have more than one version of a given C++ toolset (e.g. configured in `user-config.jam`, or autodetected, as happens with msvc), you can request the specific version by passing *`toolset-version`* as the value of the `toolset` feature, for example `toolset=msvc-8.0`.

If a feature has a fixed set of values it can be specified more than once on the command line. In which case, everything will be built several times -- once for each specified value of a feature. For example, if you use

```
bjam -link=static -link=shared -threading=single -threading=multi
```

Then a total of 4 builds will be performed. For convenience, instead of specifying all requested values of a feature in separate command line elements, you can separate the values with commands, for example:

```
bjam -link=static,shared -threading=single,multi
```

The comma has special meaning only if the feature has a fixed set of values, so

```
bjam -include=static,shared
```

is not treated specially.

Targets

All command line elements that are neither options nor properties are the names of the targets to build. See the section called "Target identifiers and references". If no target is specified, the project in the current directory is built.

Declaring Targets

A *Main target* is a user-defined named entity that can be built, for example an executable file. Declaring a main target is usually done using one of the main target rules described in the section called "Builtin rules". The user can also declare custom main target rules as shown in the section called "Main target rules".

Most main target rules in Boost.Build have the same common signature:

```
rule -rule-name -(
 - - - - -main-target-name -:
 - - - - -sources -+ -:
 - - - - -requirements -* -:
 - - - - -default-build -* -:
 - - - - -usage-requirements -* -)
```

- *`main-target-name`* is the name used to request the target on command line and to use it from other main targets. A main target name may contain alphanumeric characters, dashes ('-'), and underscores ('_').
- *`sources`* is the list of source files and other main targets that must be combined.
- *`requirements`* is the list of properties that must always be present when this main target is built.
- *`default-build`* is the list of properties that will be used unless some other value of the same feature is already specified, e.g. on the command line or by propagation from a dependent target.
- *`usage-requirements`* is the list of properties that will be propagated to all main targets that use this one, i.e. to all its dependents.

Some main target rules have a different list of parameters as explicitly stated in their documentation.

The actual requirements for a target are obtained by refining requirements of the project where a target is declared with the explicitly specified requirements. The same is true for usage-requirements. More details can be found in the section called “Property refinement”

Name

The name of main target has two purposes. First, it's used to refer to this target from other targets and from command line. Second, it's used to compute the names of the generated files. Typically, filenames are obtained from main target name by appending system-dependent suffixes and prefixes.

The name of a main target can contain alphanumeric characters, dashes, undescores and dots. The entire name is significant when resolving references from other targets. For determining filenames, only the part before the first dot is taken. For example:

```
obj -test.release -: -test.cpp -: -<variant>release -;
obj -test.debug -: -test.cpp -: -<variant>debug -;
```

will generate two files named `test.obj` (in two different directories), not two files named `test.release.obj` and `test.debug.obj`.

Sources

The list of sources specifies what should be processed to get the resulting targets. Most of the time, it's just a list of files. Sometimes, you'll want to automatically construct the list of source files rather than having to spell it out manually, in which case you can use the `glob` rule. Here are two examples:

```
exe -a -: -a.cpp -; - - - - - - - - - - -# -a.cpp -is -the -only -source -file
exe -b -: -[ -glob -*.cpp -] -; - -# -all -.cpp -files -in -this -directory -are -sources
```

Unless you specify a file with an absolute path, the name is considered relative to the source directory—which is typically the directory where the Jamfile is located, but can be changed as described in the section called “Projects” [23]

The list of sources can also refer to other main targets. Targets in the same project can be referred to by name, while targets in other projects must be qualified with a directory or a symbolic project name. The directory/project name is separated from the target name by a double forward slash. There is no special syntax to distinguish the directory name from the project name—the part before the double slash is first looked up as project name, and then as directory name. For example:

```
lib -helper -: -helper.cpp -;
exe -a -: -a.cpp -helper -;
# -Since -all -project -ids -start -with -slash, -".." -is -a -directory -name.
exe -b -: -b.cpp -..//utils -;
exe -c -: -c.cpp -/boost/program_options//program_options -;
```

The first exe uses the library defined in the same project. The second one uses some target (most likely a library) defined by a Jamfile one level higher. Finally, the third target uses a C++ Boost library, referring to it using its absolute symbolic name. More information about target references can be found in the section called “Dependent Targets” and the section called “Target identifiers and references”.

Requirements

Requirements are the properties that should always be present when building a target. Typically, they are includes and defines:

```
exe -hello -: -hello.cpp -: -<include>/opt/boost -<define>MY_DEBUG -;
```

```
```

There is a number of other features, listed in the section called “Builtin features”. For example if a library can only be built statically, or a file can't be compiled with optimization due to a compiler bug, one can use

```
lib -util -: -util.cpp -: -<link>static -;
obj -main -: -main.cpp -: -<optimization>off -;
```

Sometimes, particular relationships need to be maintained among a target's build properties. This can be achieved with *conditional requirements*. For example, you might want to set specific `#defines` when a library is built as shared, or when a target's `release` variant is built in release mode.

```
lib -network -: -network.cpp
 - - - -: -<link>shared:<define>NEWORK_LIB_SHARED
 - - - - -<variant>release:<define>EXTRA_FAST
 - - - -;
```

In the example above, whenever `network` is built with `<link>shared`, `<define>NEWORK_LIB_SHARED` will be in its properties, too.

You can use several properties in the condition, for example:

```
lib -network -: -network.cpp
 - - - -: -<toolset>gcc,<optimization>speed:<define>USE_INLINE_ASSEMBLER
 - - - -;
```

A more powerful variant of conditional requirements is *indirect conditional requirements*. You can provide a rule that will be called with the current build properties and can compute additional properties to be added. For example:

```
lib -network -: -network.cpp
 - - - -: -<conditional>@my-rule
 - - - -;
rule -my-rule -( -properties -* -)
{
 - - - -local -result -;
 - - - -if -<toolset>gcc -<optimization>speed -in -$(properties)
 - - - -{
 - - - - - - - -result -+= -<define>USE_INLINE_ASSEMBLER -;
 - - - -}
 - - - -return -$(result) -;
}
```

This example is equivalent to the previous one, but for complex cases, indirect conditional requirements can be easier to write and understand.

Requirements explicitly specified for a target are usually combined with the requirements specified for the containing project. You can cause a target to completely ignore specific project's requirement using the syntax by adding a minus sign before a property, for example:

```
exe -main -: -main.cpp -: --<define>UNNECESSARY_DEFINE -;
```

This syntax is the only way to ignore free properties from a parent, such as defines. It can be also useful for ordinary properties. Consider this example:

```
project -test -: -requirements -<threading>multi -;
exe -test1 -: -test1.cpp -;
```

```
exe -test2 -: -test2.cpp -: -<threading>single -;
exe -test3 -: -test3.cpp -: --<threading>multi -;
```

Here, `test1` inherits project requirements and will always be built in multi-threaded mode. The `test2` target *overrides* project's requirements and will always be built in single-threaded mode. In contrast, the `test3` target *removes* a property from project requirements and will be built either in single-threaded or multi-threaded mode depending on which variant is requested by the user.

Note that the removal of requirements is completely textual: you need to specify exactly the same property to remove it.

Default Build

The `default-build` parameter is a set of properties to be used if the build request does not otherwise specify a value for features in the set. For example:

```
exe -hello -: -hello.cpp -: -: -<threading>multi -;
```

would build a multi-threaded target unless the user explicitly requests a single-threaded version. The difference between requirements and default-build is that requirements cannot be overridden in any way.

Additional Information

The ways a target is built can be so different that describing them using conditional requirements would be hard. For example, imagine that a library actually uses different source files depending on the toolset used to build it. We can express this situation using *target alternatives*:

```
lib -demangler -: -dummy_demangler.cpp -; - - - - - - - - - - - - - - - - - - - - - -# -altern
ative -1
lib -demangler -: -demangler_gcc.cpp -: -<toolset>gcc -; - - -# -alternative -2
lib -demangler -: -demangler_msvc.cpp -: -<toolset>msvc -; -# -alternative -3
```

In the example above, when built with `gcc` or `msvc`, `demangler` will use a source file specific to the toolset. Otherwise, it will use a generic source file, `dummy_demangler.cpp`.

It is possible to declare a target inline, i.e. the "sources" parameter may include calls to other main rules. For example:

```
exe -hello -: -hello.cpp
 - - - -[ -obj -helpers -: -helpers.cpp -: -<optimization>off -] -;
```

Will cause "helpers.cpp" to be always compiled without optimization. When referring to an inline main target, its declared name must be prefixed by its parent target's name and two dots. In the example above, to build only helpers, one should run `bjam hello..helpers`.

When no target is requested on the command line, all targets in the current project will be built. If a target should be built only by explicit request, this can be expressed by the `explicit` rule:

```
explicit -install_programs -;
```

Projects

As mentioned before, targets are grouped into projects, and each Jamfile is a separate project. Projects are useful because they allow us to group related targets together, define properties common to all those targets, and assign a symbolic name to the project that can be used in referring to its targets.

Projects are named using the `project` rule, which has the following syntax:

```
project -id -: -attributes -;
```

Here, *`attributes`* is a sequence of rule arguments, each of which begins with an attribute-name and is followed by any number of build properties. The list of attribute names along with its handling is also shown in the table below. For example, it is possible to write:

```
project -tennis
 - - - -: -requirements -<threading>multi
 - - - -: -default-build -release
 - - - -;
```

The possible attributes are listed below.

Project id is a short way to denote a project, as opposed to the Jamfile's pathname. It is a hierarchical path, unrelated to filesystem, such as "boost/thread". Target references make use of project ids to specify a target.

Source location specifies the directory where sources for the project are located.

Project requirements are requirements that apply to all the targets in the projects as well as all subprojects.

Default build is the build request that should be used when no build request is specified explicitly.

The default values for those attributes are given in the table below.

Table 3.

Attribute	**Name**	**Default value**	**Handling by the `project` rule**
Project id	none	none	Assigned from the first parameter of the 'project' rule. It is assumed to denote absolute project id.
Source location	`source-location`	The location of jamfile for the project	Sets to the passed value
Requirements	`requirements`	The parent's requirements	The parent's requirements are refined with the passed requirement and the result is used as the project requirements.
Default build	`default-build`	none	Sets to the passed value
Build directory	`build-dir`	Empty if the parent has no build directory set. Otherwise, the parent's build directory with the relative path from parent to the current project appended to it.	Sets to the passed value, interpreted as relative to the project's location.

Besides defining projects and main targets, Jamfiles often invoke various utility rules. For the full list of rules that can be directly used in Jamfile see the section called “Builtin rules”.

Each subproject inherits attributes, constants and rules from its parent project, which is defined by the nearest Jamfile in an ancestor directory above the subproject. The top-level project is declared in a file called `Jamroot` rather than `Jamfile`. When loading a project, Boost.Build looks for either `Jamroot` or `Jamfile`. They are handled identically, except that if the file is called `Jamroot`, the search for a parent project is not performed.

Even when building in a subproject directory, parent project files are always loaded before those of their subprojects, so that every definition made in a parent project is always available to its children. The loading order of any other projects is unspecified. Even if one project refers to another via the `use-project` or a target reference, no specific order should be assumed.

Note

Giving the root project the special name "`Jamroot`" ensures that Boost.Build won't misinterpret a directory above it as the project root just because the directory contains a Jamfile.

The Build Process

When you've described your targets, you want Boost.Build to run the right tools and create the needed targets. This section will describe two things: how you specify what to build, and how the main targets are actually constructed.

The most important thing to note is that in Boost.Build, unlike other build tools, the targets you declare do not correspond to specific files. What you declare in a Jamfile is more like a "metatarget." Depending on the properties you specify on the command line, each metatarget will produce a set of real targets corresponding to the requested properties. It is quite possible that the same metatarget is built several times with different properties, producing different files.

Tip

This means that for Boost.Build, you cannot directly obtain a build variant from a Jamfile. There could be several variants requested by the user, and each target can be built with different properties.

Build Request

The command line specifies which targets to build and with which properties. For example:

```
bjam -app1 -lib1//lib1 -toolset=gcc -variant=debug -optimization=full
```

would build two targets, "app1" and "lib1//lib1" with the specified properties. You can refer to any targets, using target id and specify arbitrary properties. Some of the properties are very common, and for them the name of the property can be omitted. For example, the above can be written as:

```
bjam -app1 -lib1//lib1 -gcc -debug -optimization=full
```

The complete syntax, which has some additional shortcuts, is described in the section called "Invocation".

Building a main target

When you request, directly or indirectly, a build of a main target with specific requirements, the following steps are done. Some brief explanation is provided, and more details are given in the section called "Build process".

1. Applying default build. If the default-build property of a target specifies a value of a feature that is not present in the build request, that value is added.

2. Selecting the main target alternative to use. For each alternative we look how many properties are present both in alternat-

ive's requirements, and in build request. The alternative with large number of matching properties is selected.

3. Determining "common" properties. The build request is refined with target's requirements. The conditional properties in requirements are handled as well. Finally, default values of features are added.

4. Building targets referred by the sources list and dependency properties. The list of sources and the properties can refer to other target using target references. For each reference, we take all propagated properties, refine them by explicit properties specified in the target reference, and pass the resulting properties as build request to the other target.

5. Adding the usage requirements produced when building dependencies to the "common" properties. When dependencies are built in the previous step, they return both the set of created "real" targets, and usage requirements. The usage requirements are added to the common properties and the resulting property set will be used for building the current target.

6. Building the target using generators. To convert the sources to the desired type, Boost.Build uses "generators" --- objects that correspond to tools like compilers and linkers. Each generator declares what type of targets it can produce and what type of sources it requires. Using this information, Boost.Build determines which generators must be run to produce a specific target from specific sources. When generators are run, they return the "real" targets.

7. Computing the usage requirements to be returned. The conditional properties in usage requirements are expanded and the result is returned.

Building a Project

Often, a user builds a complete project, not just one main target. In fact, invoking **bjam** without arguments builds the project defined in the current directory.

When a project is built, the build request is passed without modification to all main targets in that project. It's is possible to prevent implicit building of a target in a project with the `explicit` rule:

```
explicit -hello_test -;
```

would cause the `hello_test` target to be built only if explicitly requested by the user or by some other target.

The Jamfile for a project can include a number of `build-project` rule calls that specify additional projects to be built.

Common tasks

This section describes main targets types that Boost.Build supports out-of-the-box. Unless otherwise noted, all mentioned main target rules have the common signature, described in the section called “Declaring Targets”.

Programs

Programs are created using the `exe` rule, which follows the common syntax. For example:

```
exe -hello -: -hello.cpp -some_library.lib -/some_project//library
 - - - - - - - - - -: -<threading>multi
 - - - - - - - - - -;
```

This will create an executable file from the sources -- in this case, one C++ file, one library file present in the same directory, and another library that is created by Boost.Build. Generally, sources can include C and C++ files, object files and libraries. Boost.Build will automatically try to convert targets of other types.

> **Tip**
>
> On Windows, if an application uses shared libraries, and both the application and the libraries are built using Boost.Build, it is not possible to immediately run the application, because the `PATH` environment variable should include the path to the libraries. It means you have to either add the paths manually, or have the build place the application and the libraries into the same directory. See the section called “Installing”.

Libraries

Library targets are created using the `lib` rule, which follows the common syntax. For example:

```
lib -helpers -: -helpers.cpp -;
```

This will define a library target named `helpers` built from the `helpers.cpp` source file.

Depending on the given <link> feature value the library will be either static or shared.

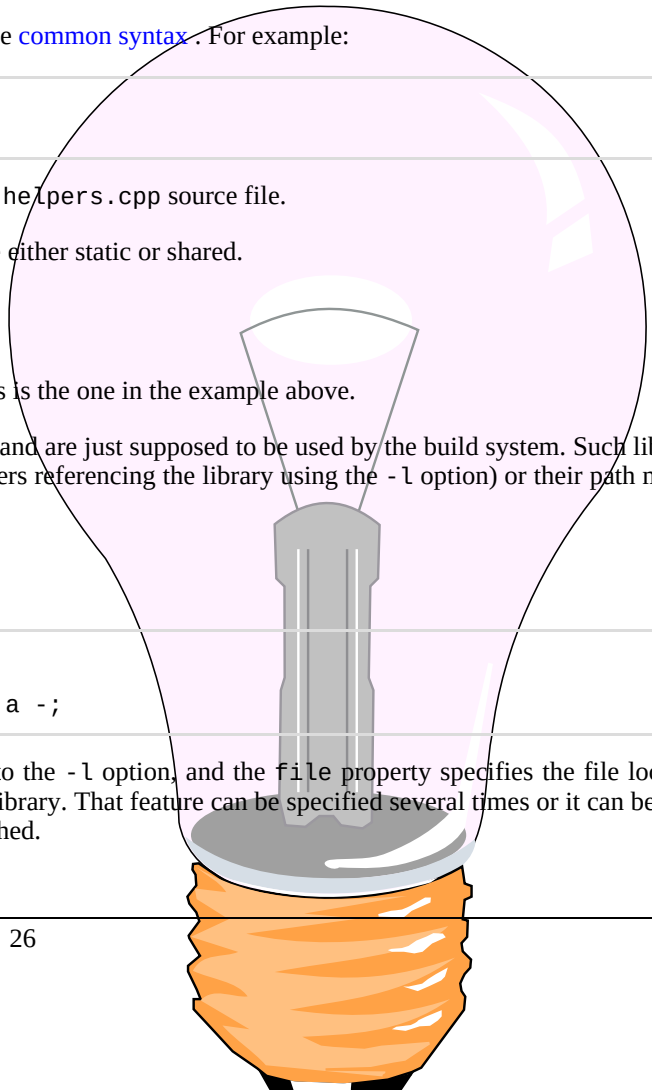
Library targets may be used to represent:

- `Built libraries` that get built from specified sources, as is the one in the example above.
- `Prebuilt libraries` which already exist on the system and are just supposed to be used by the build system. Such libraries may be searched for by the tools using them (typically linkers referencing the library using the `-l` option) or their path may be known in advance by the build system.

The syntax for these case is given below:

```
lib -z -: -: -<name>z -<search>/home/ghost -;
lib -compress -: -: -<file>/opt/libs/compress.a -;
```

The `name` property specifies the name that should be passed to the `-l` option, and the `file` property specifies the file location. The `search` feature specifies paths in which to search for the library. That feature can be specified several times or it can be omitted, in which case only the default compiler paths will be searched.



The difference between using the `file` feature as opposed to the `name` feature together with the `search` feature is that `file` is more precise. A specific file will be used as opposed to the `search` feature only adding a library path, or the `name` feature giving only the basic name of the library. The search rules are specific to the linker used. For example, given these definition:

```
lib -a -: -: -<variant>release -<file>/pool/release/a.so -;
lib -a -: -: -<variant>debug -<file>/pool/debug/a.so -;
lib -b -: -: -<variant>release -<file>/pool/release/b.so -;
lib -b -: -: -<variant>debug -<file>/pool/debug/b.so -;
```

It is possible to use a release version of `a` and debug version of `b`. Had we used the `name` and `search` features, the linker would have always picked either the release or the debug versions.

For convenience, the following syntax is allowed:

```
lib -z -;
lib -gui -db -aux -;
```

which has exactly the same effect as:

```
lib -z -: -: -<name>z -;
lib -gui -: -: -<name>gui -;
lib -db -: -: -<name>db -;
lib -aux -: -: -<name>aux -;
```

When a library references another library you should put that other library in its list of sources. This will do the right thing in all cases. For portability, you should specify library dependencies even for searched and prebuilt libraries, othewise, static linking on Unix will not work. For example:

```
lib -z -;
lib -png -: -z -: -<name>png -;
```

Note

When a library has a shared library defined as its source, or a static library has another static library defined as its source then any target linking to the first library with automatically link to its source library as well.

On the other hand, when a shared library has a static library defined as its source then the first library will be built so that it completely includes the second one.

If you do not want shared libraries to include all libraries specified in its sources (especially statically linked ones), you would need to use the following:

```
lib -b -: -a.cpp -;
lib -a -: -a.cpp -: -<use>b -: -: -<library>b -;
```

This specifies that library `a` uses library `b`, and causes all executables that link to `a` also link to `b`. In this case, even for shared linking, the `a` library will not even refer to `b`.

One Boost.Build feature that is often very useful for defining library targets are usage requirements. For example, imagine that you want you build a `helpers` library and its interface is described in its `helpers.hpp` header file located in the same directory as the `helpers.cpp` source file. Then you could add the following to the Jamfile located in that same directory:

```
lib -helpers -: -helpers.cpp -: -: -: -<include>. -;
```

which would automatically add the directory where the target has been defined (and where the library's header file is located) to the compiler's include path for all targets using the `helpers` library. This feature greatly simplifies Jamfiles.

Alias

The `alias` rule gives an alternative name to a group of targets. For example, to give the name `core` to a group of three other targets with the following code:

```
alias -core -: -im -reader -writer -;
```

Using `core` on the command line, or in the source list of any other target is the same as explicitly using `im` , `reader`, and `writer`.

Another use of the `alias` rule is to change build properties. For example, if you want to use link statically to the Boost Threads library, you can write the following:

```
alias -threads -: -/boost/thread//boost_thread -: -<link>static -;
```

and use only the `threads` alias in your Jamfiles.

You can also specify usage requirements for the `alias` target. If you write the following:

```
alias -header_only_library -: -: -: -: - -<include>/usr/include/header_only_library -;
```

then using `header_only_library` in sources will only add an include path. Also note that when an alias has sources, their usage requirements are propagated as well. For example:

```
lib -library1 -: -library1.cpp -: -: -: -<include>/library/include1 -;
lib -library2 -: -library2.cpp -: -: -: -<include>/library/include2 -;
alias -static_libraries -: -library1 -library2 -: -<link>static -;
exe -main -: -main.cpp -static_libraries -;
```

will compile `main.cpp` with additional includes required for using the specified static libraries.

Installing

This section describes various ways to install built target and arbitrary files.

Basic install

For installing a built target you should use the `install` rule, which follows the common syntax. For example:

```
install -dist -: -hello -helpers -;
```

will cause the targets `hello` and `helpers` to be moved to the `dist` directory, relative to the Jamfile's directory. The directory can be changed using the `location` property:

```
install -dist -: -hello -helpers -: -<location>/usr/bin -;
```

While you can achieve the same effect by changing the target name to `/usr/bin`, using the `location` property is better as it allows you to use a mnemonic target name.

The `location` property is especially handy when the location is not fixed, but depends on the build variant or environment variables:

```
install -dist -: -hello -helpers -:
 - - - -<variant>release:<location>dist/release
 - - - -<variant>debug:<location>dist/debug -;
install -dist2 -: -hello -helpers -: -<location>$(DIST) -;
```

See also conditional properties and environment variables

Installing with all dependencies

Specifying the names of all libraries to install can be boring. The `install` allows you to specify only the top-level executable targets to install, and automatically install all dependencies:

```
install -dist -: -hello
 - - - - - - - - - - -: -<install-dependencies>on -<install-type>EXE
 - - - - - - - - - - - - -<install-type>LIB
 - - - - - - - - - - -;
```

will find all targets that `hello` depends on, and install all of those which are either executables or libraries. More specifically, for each target, other targets that were specified as sources or as dependency properties, will be recursively found. One exception is that targets referred with the `use` feature are not considered, as that feature is typically used to refer to header-only libraries. If the set of target types is specified, only targets of that type will be installed, otherwise, all found target will be installed.

Preserving Directory Hierarchy

By default, the `install` rule will strip paths from its sources. So, if sources include `a/b/c.hpp`, the `a/b` part will be ignored. To make the `install` rule preserve the directory hierarchy you need to use the `<install-source-root>` feature to specify the root of the hierarchy you are installing. Relative paths from that root will be preserved. For example, if you write:

```
install -headers
 - - - -: -a/b/c.h
 - - - -: -<location>/tmp -<install-source-root>a
 - - - -;
```

the a file named `/tmp/b/c.h` will be created.

The glob-tree rule can be used to find all files below a given directory, making it easy to install an entire directory tree.

Installing into Several Directories

The `alias` rule can be used when targets need to be installed into several directories:

```
alias -install -: -install-bin -install-lib -;
install -install-bin -: -applications -: -/usr/bin -;
install -install-lib -: -helper -: -/usr/lib -;
```

Because the `install` rule just copies targets, most free features [1] have no effect when used in requirements of the `install` rule. The only two that matter are `dependency` and, on Unix, `dll-path` .

[1] see the definition of "free" in the section called “Feature Attributes”.

Note

(Unix specific) On Unix, executables built using Boost.Build typically contain the list of paths to all used shared libraries. For installing, this is not desired, so Boost.Build relinks the executable with an empty list of paths. You can also specify additional paths for installed executables using the `dll-path` feature.

Testing

Boost.Build has convenient support for running unit tests. The simplest way is the `unit-test` rule, which follows the common syntax. For example:

```
unit-test -helpers_test -: -helpers_test.cpp -helpers -;
```

The `unit-test` rule behaves like the `exe` rule, but after the executable is created it is also run. If the executable returns an error code, the build system will also return an error and will try running the executable on the next invocation until it runs successfully. This behaviour ensures that you can not miss a unit test failure.

By default, the executable is run directly. Sometimes, it is desirable to run the executable using some helper command. You should use the `testing.launcher` property to specify the name of the helper command. For example, if you write:

```
unit-test -helpers_test
 - - -: -helpers_test.cpp -helpers
 - - -: -<testing.launcher>valgrind
 - - -;
```

The command used to run the executable will be:

```
valgrind -bin/$toolset/debug/helpers_test
```

There are few specialized testing rules, listed below:

```
rule -compile -( -sources -: -requirements -* -: -target-name -? -)
rule -compile-fail -( -sources -: -requirements -* -: -target-name -? -)
rule -link -( -sources -+ -: -requirements -* -: -target-name -? -)
rule -link-fail -( -sources -+ -: -requirements -* -: -target-name -? -)
```

They are given a list of sources and requirements. If the target name is not provided, the name of the first source file is used instead. The `compile*` tests try to compile the passed source. The `link*` rules try to compile and link an application from all the passed sources. The `compile` and `link` rules expect that compilation/linking succeeds. The `compile-fail` and `link-fail` rules expect that the compilation/linking fails.

There are two specialized rules for running applications, which are more powerful than the `unit-test` rule. The `run` rule has the following signature:

```
rule -run -( -sources -+ -: -args -* -: -input-files -* -: -requirements -* -: -target-name -?
 - - - -: -default-build -* -)
```

The rule builds application from the provided sources and runs it, passing `args` and `input-files` as command-line arguments. The `args` parameter is passed verbatim and the values of the `input-files` parameter are treated as paths relative to containing Jamfile, and are adjusted if **bjam** is invoked from a different directory. The `run-fail` rule is identical to the `run` rule, except that it expects that the run fails.

All rules described in this section, if executed successfully, create a special manifest file to indicate that the test passed. For the `unit-test` rule the files is named *`target-name`*`.passed` and for the other rules it is called *`target-name`*`.test`. The `run*` rules also capture all output from the program, and store it in a file named *`target-name`*`.output`.

If the `preserve-test-targets` feature has the value `off`, then `run` and the `run-fail` rules will remove the executable after running it. This somewhat decreases disk space requirements for continuous testing environments. The default value of `preserve-test-targets` feature is `on`.

It is possible to print the list of all test targets (except for `unit-test`) declared in your project, by passing the `--dump-tests` command-line option. The output will consist of lines of the form:

```
boost-test(test-type) -path -: -sources
```

It is possible to process the list of tests, the output of bjam during command run, and the presense/absense of the `*.test` files created when test passes into human-readable status table of tests. Such processing utilities are not included in Boost.Build.

Custom commands

When you use most of main target rules, Boost.Build automatically figures what commands to run and it what order. As soon as you want to use new file types or support new tools, one approach is to extend Boost.Build to smoothly support them, as documented in *Extender Manual*. However, if there is only a single place where the new tool is used, it might be easier to just explicitly specify the commands to run.

Three main target rules can be used for that. The `make` rule allows you to construct a single file from any number of source file, by running a command you specify. The `notfile` rule allows you to run an arbitrary command, without creating any files. And finaly, the `generate` rule allows you to describe transformation using Boost.Build's virtual targets. This is higher-level than file names that the `make` rule operates with and allows you to create more than one target, create differently named targets depending on properties or use more than one tool.

The `make` rule is used when you want to create one file from a number of sources using some specific command. The `notfile` is used to unconditionally run a command.

Suppose you want to create file `file.out` from file `file.in` by running command **in2out**. Here is how you would do this in Boost.Build:

```
make -file.out -: -file.in -: -@in2out -;
actions -in2out
{
 - - - -in2out -$(<) -$(>)
}
```

If you run **bjam** and `file.out` does not exist, Boost.Build will run the **in2out** command to create that file. For more details on specifying actions, see the section called “Boost.Jam Language” [14].

It could be that you just want to run some command unconditionally, and that command does not create any specific files. For that you can use the `notfile` rule. For example:

```
notfile -echo_something -: -@echo -;
actions -echo
{
 - - - -echo -"something"
}
```

The only difference from the `make` rule is that the name of the target is not considered a name of a file, so Boost.Build will unconditionally run the action.

The `generate` rule is used when you want to express transformations using Boost.Build's virtual targets, as opposed to just filenames. The `generate` rule has the standard main target rule signature, but you are required to specify the `generating-rule`

property. The value of the property should be in the form `@rule-name`, the named rule should have the following signature:

```
rule -generating-rule -( -project -name -: -property-set -: -sources -* -)
```

and will be called with an instance of the `project-target` class, the name of the main target, an instance of the `property-set` class containing build properties, and the list of instances of the `virtual-target` class corresponding to sources. The rule must return a list of `virtual-target` instances. The interface of the `virtual-target` class can be learned by looking at the `build/virtual-target.jam` file. The `generate` example contained in the Boost.Build distribution illustrates how the `generate` rule can be used.

Precompiled Headers

Precompiled headers is a mechanism to speed up compilation by creating a partially processed version of some header files, and then using that version during compilations rather then repeatedly parsing the original headers. Boost.Build supports precompiled headers with gcc and msvc toolsets.

To use precompiled headers, follow the following steps:

1. Create a header that includes headers used by your project that you want precompiled. It is better to include only headers that are sufficiently stable — like headers from the compiler and external libraries. Please wrap the header in `#ifdef BOOST_BUILD_PCH_ENABLED`, so that the potentially expensive inclusion of headers is not done when PCH is not enabled. Include the new header at the top of your source files.

2. Declare a new Boost.Build target for the precompiled header and add that precompiled header to the sources of the target whose compilation you want to speed up:

```
cpp-pch -pch -: -pch.hpp -;
exe -main -: -main.cpp -pch -;
```

   You can use the `c-pch` rule if you want to use the precompiled header in C programs.

The `pch` example in Boost.Build distribution can be used as reference.

Please note the following:

- The inclusion of the precompiled header must be the first thing in a source file, before any code or preprocessor directives.
- The build properties used to compile the source files and the precompiled header must be the same. Consider using project requirements to assure this.
- Precompiled headers must be used purely as a way to improve compilation time, not to save the number of `#include` statements. If a source file needs to include some header, explicitly include it in the source file, even if the same header is included from the precompiled header. This makes sure that your project will build even if precompiled headers are not supported.
- On the gcc compiler, the name of the header being precompiled must be equal to the name of the `cpp-pch` target. This is a gcc requirement.
- Prior to version 4.2, the gcc compiler did not allow anonymous namespaces in precompiled headers, which limits their utility. See the bug report for details.

Generated headers

Usually, Boost.Build handles implicit dependendies completely automatically. For example, for C++ files, all `#include` statements are found and handled. The only aspect where user help might be needed is implicit dependency on generated files.

By default, Boost.Build handles such dependencies within one main target. For example, assume that main target "app" has two

sources, "app.cpp" and "parser.y". The latter source is converted into "parser.c" and "parser.h". Then, if "app.cpp" includes "parser.h", Boost.Build will detect this dependency. Moreover, since "parser.h" will be generated into a build directory, the path to that directory will automatically added to include path.

Making this mechanism work across main target boundaries is possible, but imposes certain overhead. For that reason, if there is implicit dependency on files from other main targets, the `<implicit-dependency>` [link] feature must be used, for example:

```
lib -parser -: -parser.y -;
exe -app -: -app.cpp -: -<implicit-dependency>parser -;
```

The above example tells the build system that when scanning all sources of "app" for implicit-dependencies, it should consider targets from "parser" as potential dependencies.

Cross-compilation

Boost.Build supports cross compilation with the gcc and msvc toolsets.

When using gcc, you first need to specify your cross compiler in `user-config.jam` (see the section called “Configuration”), for example:

```
using -gcc -: -arm -: -arm-none-linux-gnueabi-g++ -;
```

After that, if the host and target os are the same, for example Linux, you can just request that this compiler version to be used:

```
bjam -toolset=gcc-arm
```

If you want to target different operating system from the host, you need to additionally specify the value for the `target-os` feature, for example:

```
# -On -windows -box
bjam -toolset=gcc-arm -target-os=linux
# -On -Linux -box
bjam -toolset=gcc-mingw -target-os=windows
```

For the complete list of allowed opeating system names, please see the documentation for target-os feature.

When using the msvc compiler, it's only possible to cross-compiler to a 64-bit system on a 32-bit host. Please see the section called “64-bit support” for details.

Reference

General information

Initialization

bjam's first job upon startup is to load the Jam code that implements the build system. To do this, it searches for a file called `boost-build.jam`, first in the invocation directory, then in its parent and so forth up to the filesystem root, and finally in the directories specified by the environment variable BOOST_BUILD_PATH. When found, the file is interpreted, and should specify the build system location by calling the boost-build rule:

```
rule -boost-build -( -location -? -)
```

If location is a relative path, it is treated as relative to the directory of `boost-build.jam`. The directory specified by that location and the directories in BOOST_BUILD_PATH are then searched for a file called `bootstrap.jam`, which is expected to bootstrap the build system. This arrangement allows the build system to work without any command-line or environment variable settings. For example, if the build system files were located in a directory "build-system/" at your project root, you might place a `boost-build.jam` at the project root containing:

```
boost-build -build-system -;
```

In this case, running bjam anywhere in the project tree will automatically find the build system.

The default `bootstrap.jam`, after loading some standard definitions, loads two `site-config.jam` and `user-config.jam`.

Builtin rules

This section contains the list of all rules that can be used in Jamfile—both rules that define new targets and auxiliary rules.

`exe`	Creates an executable file. See the section called “Programs”.
`lib`	Creates an library file. See the section called “Libraries”.
`install`	Installs built targets and other files. See the section called “Installing”.
`alias`	Creates an alias for other targets. See the section called “Alias”.
`unit-test`	Creates an executable that will be automatically run. See the section called “Testing”.
`compile`, `compile-fail`, `link`, `link-fail`, `run`, `run-fail`	Specialized rules for testing. See the section called “Testing”.
`obj`	Creates an object file. Useful when a single source file must be compiled with special properties.
`glob`	The `glob` rule takes a list shell pattern and returns the list of files in the project's source directory that match the pattern. For example:

```
lib -tools -: -[ -glob -*.cpp -] -;
 - - - - - - - -
```

It is possible to also pass a second argument—the list of exclude patterns. The result will then include the list of files patching any of include patterns, and not matching any of the exclude patterns. For example:

```

```

```
lib -tools -: -[ -glob -*.cpp -: -file_to_exclude.cpp -bad*.cpp -] -;
 - - - - - - - -
```

`glob-tree` The `glob-tree` is similar to the `glob` except that it operates recursively from the directory of the containing Jamfile. For example:

```
ECHO -[ -glob-tree -*.cpp -: -.svn -] -;
 - - - - - - - -
```

will print the names of all C++ files in your project. The `.svn` exclude pattern prevents the `glob-tree` rule from entering administrative directories of the Subversion version control system.

`project` Declares project id and attributes, including project requirements. See the section called “Projects”.

`use-project` Assigns a symbolic project ID to a project at a given path. This rule must be better documented!

`explicit` The `explicit` rule takes a single parameter—a list of target names. The named targets will be marked explicit, and will be built only if they are explicitly requested on the command line, or if their dependents are built. Compare this to ordinary targets, that are built implicitly when their containing project is built.

`constant` Sets project-wide constant. Takes two parameters: variable name and a value and makes the specified variable name accessible in this Jamfile and any child Jamfiles. For example:

```
constant -VERSION -: -1.34.0 -;
 - - - - - - - -
```

`path-constant` Same as `constant` except that the value is treated as path relative to Jamfile location. For example, if **bjam** is invoked in the current directory, and Jamfile in `helper` subdirectory has:

```
path-constant -DATA -: -data/a.txt -;
 - - - - - - - -
```

then the variable `DATA` will be set to `helper/data/a.txt`, and if **bjam** is invoked from the `helper` directory, then the variable `DATA` will be set to `data/a.txt`.

`build-project` Cause some other project to be built. This rule takes a single parameter—a directory name relative to the containing Jamfile. When the containing Jamfile is built, the project located at that directory will be built as well. At the moment, the parameter to this rule should be a directory name. Project ID or general target references are not allowed.

`test-suite` This rule is deprecated and equivalent to `alias`.

Builtin features

This section documents the features that are built-in into Boost.Build. For features with a fixed set of values, that set is provided, with the default value listed first.

`variant` A feature combining several low-level features, making it easy to request common build configurations.

Allowed values: `debug`, `release`, `profile`.

The value `debug` expands to

```
<optimization>off -<debug-symbols>on -<inlining>off -<runtime-debugging>on
```

The value `release` expands to

```
<optimization>speed -<debug-symbols>off -<inlining>full -<runtime-debugging>off
```

The value `profile` expands to the same as `release`, plus:

```
<profiling>on -<debug-symbols>on
```

Users can define their own build variants using the `variant` rule from the `common` module.

Note: Runtime debugging is on in debug builds to suit the expectations of people used to various IDEs.

`link` **Allowed values:** `shared`, `static`

A feature controling how libraries are built.

`runtime-link` **Allowed values:** `shared`, `static`

Controls if a static or shared C/C++ runtime should be used. There are some restrictions how this feature can be used, for example on some compilers an application using static runtime should not use shared libraries at all, and on some compilers, mixing static and shared runtime requires extreme care. Check your compiler documentation for more details.

`threading` **Allowed values:** `single`, `multi`

Controls if the project should be built in multi-threaded mode. This feature does not necessary change code generation in the compiler, but it causes the compiler to link to additional or different runtime libraries, and define additional preprocessor symbols (for example, `_MT` on Windows and `_REENTRANT` on Linux). How those symbols affect the compiled code depends on the code itself.

`source` The `<source>X` feature has the same effect on building a target as putting X in the list of sources. It is useful when you want to add the same source to all targets in the project (you can put <source> in requirements) or to conditionally include a source (using conditional requirements, see the section called “Conditions and alternatives”). See also the `<library>` feature.

`library` This feature is almost equivalent to the `<source>` feature, except that it takes effect only for linking. When you want to link all targets in a Jamfile to certain library, the `<library>` feature is preferred over `<source>X` -- the latter will add the library to all targets, even those that have nothing to do with libraries.

`dependency` Introduces a dependency on the target named by the value of this feature (so it will be brought up-to-date whenever the target being declared is). The dependency is not used in any other way.

`use` Introduces a dependency on the target named by the value of this feature (so it will be brought up-to-date whenever the target being declared is), and adds its usage requirements to the build properties of the target being declared. The dependency is not used in any other way. The primary use case is when you want the usage requirements (such as `#include` paths) of some library to be applied, but do not want to link to it.

`dll-path` Specify an additional directory where the system should look for shared libraries when the executable or shared library is run. This feature only affects Unix compilers. Plase see the section

called " Why are the `dll-path` and `hardcode-dll-paths` properties useful? " in *Frequently Asked Questions* for details.

`hardcode-dll-paths`

Controls automatic generation of dll-path properties.

Allowed values: `true`, `false`. This property is specific to Unix systems. If an executable is built with `<hardcode-dll-paths>true`, the generated binary will contain the list of all the paths to the used shared libraries. As the result, the executable can be run without changing system paths to shared libraries or installing the libraries to system paths. This is very convenient during development. Plase see the FAQ entry for details. Note that on Mac OSX, the paths are unconditionally hardcoded by the linker, and it is not possible to disable that behaviour.

`cflags`, `cxxflags`, `linkflags`

The value of those features is passed without modification to the corresponding tools. For `cflags` that is both the C and C++ compilers, for `cxxflags` that is the C++ compiler and for `linkflags` that is the linker. The features are handy when you are trying to do something special that cannot be achieved by a higher-level feature in Boost.Build.

`include`

Specifies an additional include path that is to be passed to C and C++ compilers.

`warnings`

The `<warnings>` feature controls the warning level of compilers. It has the following values:

- `off` - disables all warnings.
- `on` - enables default warning level for the tool.
- `all` - enables all warnings.

Default value is `all`.

`warnings-as-errors`

The `<warnings-as-errors>` makes it possible to treat warnings as errors and abort compilation on a warning. The value `on` enables this behaviour. The default value is `off`.

`build`

Allowed values: `no`

The `build` feature is used to conditionally disable build of a target. If `<build>no` is in properties when building a target, build of that target is skipped. Combined with conditional requirements this allows you to skip building some target in configurations where the build is known to fail.

`tag`

The `tag` feature is used to customize the name of the generated files. The value should have the form:

```
@rulename
```

where *`rulename`* should be a name of a rule with the following signature:

```
rule -tag -( -name -: -type -? -: -property-set -)
```

The rule will be called for each target with the default name computed by Boost.Build, the type of the target, and property set. The rule can either return a string that must be used as the name of the target, or an empty string, in which case the default name will be used.

Most typical use of the `tag` feature is to encode build properties, or library version in library target names. You should take care to return non-empty string from the tag rule only for types you care about — otherwise, you might end up modifying names of object files, generated header file and other targets for which changing names does not make sense.

`debug-symbols`

Allowed values: `on`, `off`.

The `debug-symbols` feature specifies if produced object files, executables and libraries should include debug information. Typically, the value of this feature is implicitly set by the `variant` feature, but it can be explicitly specified by the user. The most common usage is to build release variant with debugging information.

`target-os`

The operating system for which the code is to be generated. The compiler you used should be the compiler for that operating system. This option causes Boost.Build to use naming conventions suitable for that operating system, and adjust build process accordingly. For example, with gcc, it controls if import libraries are produced for shared libraries or not.

The complete list of possible values for this feature is: aix, bsd, cygwin, darwin, freebsd, hpux, iphone, linux, netbsd, openbsd, osf, qnx, qnxnto, sgi, solaris, unix, unixware, windows.

See the section called “Cross-compilation” for details of crosscompilation

`architecture` The `architecture` features specifies the general processor familty to generate code for.

`instruction-set` **Allowed values:** depend on the used toolset.

The `instruction-set` specifies for which specific instruction set the code should be generated. The code in general might not run on processors with older/different instruction sets.

While Boost.Build allows a large set of possible values for this features, whether a given value works depends on which compiler you use. Please see the section called “C++ Compilers” for details.

`address-model` **Allowed values:** `32`, `64`.

The `address-model` specifies if 32-bit or 64-bit code should be generated by the compiler. Whether this feature works depends on the used compiler, its version, how the compiler is configured, and the values of the `architecture` `instruction-set` features. Please see the section called “C++ Compilers” for details.

`c++-template-depth` **Allowed values:** Any positive integer.

This feature allows configuring a C++ compiler with the maximal template instantiation depth parameter. Specific toolsets may or may not provide support for this feature depending on whether their compilers provide a corresponding command-line option.

Note: Due to some internal details in the current Boost Build implementation it is not possible to have features whose valid values are all positive integer. As a workaround a large set of allowed values has been defined for this feature and, if a different one is needed, user can easily add it by calling the feature.extend rule.

`embed-manifest`

Allowed values: on, off.

This feature is specific to the msvc toolset (see the section called “Microsoft Visual C++”), and controls whether the manifest files should be embedded inside executables and shared libraries, or placed alongside them. This feature corresponds to the IDE option found in the project settings dialog, under Configuration Properties # Manifest Tool # Input and Output # Embed manifest.

Builtin tools

Boost.Build comes with support for a large number of C++ compilers, and other tools. This section documents how to use those tools.

Before using any tool, you must declare your intention, and possibly specify additional information about the tool's configuration. This is done by calling the `using` rule, typically in your `user-config.jam`, for example:

```
using -gcc -;
```

additional parameters can be passed just like for other rules, for example:

```
using -gcc -: -4.0 -: -g++-4.0 -;
```

The options that can be passed to each tool are documented in the subsequent sections.

C++ Compilers

This section lists all Boost.Build modules that support C++ compilers and documents how each one can be initialized. The name of support module for compiler is also the value for the `toolset` feature that can be used to explicitly request that compiler.

GNU C++

The `gcc` module supports the GNU C++ compiler on Linux, a number of Unix-like system including SunOS and on Windows (either Cygwin or MinGW). On Mac OSX, it is recommended to use system gcc, see the section called “Apple Darwin gcc”.

The `gcc` module is initialized using the following syntax:

```
using -gcc -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the version is not explicitly specified, it will be automatically detected by running the compiler with the `-v` option. If the command is not specified, the **g++** binary will be searched in `PATH`.

The following options can be provided, using `<option-name>option-value` syntax:

`cflags`	Specifies additional compiler flags that will be used when compiling C sources.
`cxxflags`	Specifies additional compiler flags that will be used when compiling C++ sources.
`compileflags`	Specifies additional compiler flags that will be used when compiling both C and C++ sources.
`linkflags`	Specifies additional command line options that will be passed to the linker.
`root`	Specifies root directory of the compiler installation. This option is necessary only if it is not possible to detect this information from the compiler command—for example if the specified compiler command is a user script.
`rc`	Specifies the resource compiler command that will be used with the version of gcc that is being configured. This setting makes sense only for Windows and only if you plan to use resource files. By default **windres** will be used.
`rc-type`	Specifies the type of resource compiler. The value can be either `windres` for msvc resource compiler, or `rc` for borland's resource compiler.

In order to compile 64-bit applications, you have to specify `address-model=64`, and the `instruction-set` feature should refer to a 64 bit processor. Currently, those include `nocona`, `opteron`, `athlon64` and `athlon-fx`.

Apple Darwin gcc

The `darwin` module supports the version of gcc that is modified and provided by Apple. The configuration is essentially identical to that of the gcc module.

The darwin toolset can generate so called "fat" binaries—binaries that can run support more than one architecture, or address mode. To build a binary that can run both on Intel and PowerPC processors, specify `architecture=combined`. To build a bin-

ary that can run both in 32-bit and 64-bit modes, specify `address-model=32_64`. If you specify both of those properties, a "4-way" fat binary will be generated.

Microsoft Visual C++

The `msvc` module supports the Microsoft Visual C++ command-line tools on Microsoft Windows. The supported products and versions of command line tools are listed below:

- Visual Studio 2008—9.0
- Visual Studio 2005—8.0
- Visual Studio .NET 2003—7.1
- Visual Studio .NET—7.0
- Visual Studio 6.0, Service Pack 5—6.5

The `msvc` module is initialized using the following syntax:

```
using -msvc -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
 - - - - - - - - - -
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the version is not explicitly specified, the most recent version found in the registry will be used instead. If the special value `all` is passed as the version, all versions found in the registry will be configured. If a version is specified, but the command is not, the compiler binary will be searched in standard installation paths for that version, followed by `PATH`.

The compiler command should be specified using forward slashes, and quoted.

The following options can be provided, using `<option-name>option-value` syntax:

`cflags`	Specifies additional compiler flags that will be used when compiling C sources.
`cxxflags`	Specifies additional compiler flags that will be used when compiling C++ sources.
`compileflags`	Specifies additional compiler flags that will be used when compiling both C and C++ sources.
`linkflags`	Specifies additional command line options that will be passed to the linker.
`assembler`	The command that compiles assembler sources. If not specified, **ml** will be used. The command will be invoked after the setup script was executed and adjusted the `PATH` variable.
`compiler`	The command that compiles C and C++ sources. If not specified, **cl** will be used. The command will be invoked after the setup script was executed and adjusted the `PATH` variable.
`compiler-filter`	Command through which to pipe the output of running the compiler. For example to pass the output to STLfilt.
`idl-compiler`	The command that compiles Microsoft COM interface definition files. If not specified, **midl** will be used. The command will be invoked after the setup script was executed and adjusted the `PATH` variable.
`linker`	The command that links executables and dynamic libraries. If not specified, **link** will be used. The command will be invoked after the setup script was executed and adjusted the `PATH` variable.
`mc-compiler`	The command that compiles Microsoft message catalog files. If not specified, **mc** will be used. The command will be invoked after the setup script was executed and adjusted the `PATH` variable.

`resource-compiler`	The command that compiles resource files. If not specified, **rc** will be used. The command will be invoked after the setup script was executed and adjusted the `PATH` variable.
`setup`	The filename of the global environment setup script to run before invoking any of the tools defined in this toolset. Will not be used in case a target platform specific script has been explicitly specified for the current target platform. Used setup script will be passed the target platform identifier (x86, x86_amd64, x86_ia64, amd64 or ia64) as a arameter. If not specified a default script is chosen based on the used compiler binary, e.g. **vcvars32.bat** or **vsvars32.bat**.
`setup-amd64`, `setup-i386`, `setup-ia64`	The filename of the target platform specific environment setup script to run before invoking any of the tools defined in this toolset. If not specified the global environment setup script is used.

64-bit support

Starting with version 8.0, Microsoft Visual Studio can generate binaries for 64-bit processor, both 64-bit flavours of x86 (codenamed AMD64/EM64T), and Itanium (codenamed IA64). In addition, compilers that are itself run in 64-bit mode, for better performance, are provided. The complete list of compiler configurations are as follows (we abbreviate AMD64/EM64T to just AMD64):

- 32-bit x86 host, 32-bit x86 target
- 32-bit x86 host, 64-bit AMD64 target
- 32-bit x86 host, 64-bit IA64 target
- 64-bit AMD64 host, 64-bit AMD64 target
- 64-bit IA64 host, 64-bit IA64 target

The 32-bit host compilers can be always used, even on 64-bit Windows. On the contrary, 64-bit host compilers require both 64-bit host processor and 64-bit Windows, but can be faster. By default, only 32-bit host, 32-bit target compiler is installed, and additional compilers need to be installed explicitly.

To use 64-bit compilation you should:

1. Configure you compiler as usual. If you provide a path to the compiler explicitly, provide the path to the 32-bit compiler. If you try to specify the path to any of 64-bit compilers, configuration will not work.
2. When compiling, use `address-model=64`, to generate AMD64 code.
3. To generate IA64 code, use `architecture=ia64`

The (AMD64 host, AMD64 target) compiler will be used automatically when you are generating AMD64 code and are running 64-bit Windows on AMD64. The (IA64 host, IA64 target) compiler will never be used, since nobody has an IA64 machine to test.

It is believed that AMD64 and EM64T targets are essentially compatible. The compiler options `/favor:AMD64` and `/favor:EM64T`, which are accepted only by AMD64 targeting compilers, cause the generated code to be tuned to a specific flavor of 64-bit x86. Boost.Build will make use of those options depending on the value of the`instruction-set` feature.

Intel C++

The `intel-linux` and `intel-win` modules support the Intel C++ command-line compiler—the Linux and Windows versions respectively.

The module is initialized using the following syntax:

```
using -intel-linux -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

or

```
using -intel-win -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

respectively.

This statement may be repeated several times, if you want to configure several versions of the compiler.

If compiler command is not specified, then Boost.Build will look in `PATH` for an executable **icpc** (on Linux), or **icc.exe** (on Windows).

The following options can be provided, using `<option-name>option-value` syntax:

`cflags`	Specifies additional compiler flags that will be used when compiling C sources.
`cxxflags`	Specifies additional compiler flags that will be used when compiling C++ sources.
`compileflags`	Specifies additional compiler flags that will be used when compiling both C and C++ sources.
`linkflags`	Specifies additional command line options that will be passed to the linker.

The Linux version supports the following additional options:

`root`	Specifies root directory of the compiler installation. This option is necessary only if it is not possible to detect this information from the compiler command—for example if the specified compiler command is a user script.

HP aC++ compiler

The `acc` module supports the HP aC++ compiler for the HP-UX operating system.

The module is initialized using the following syntax:

```
using -acc -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the command is not specified, the **aCC** binary will be searched in `PATH`.

The following options can be provided, using `<option-name>option-value` syntax:

`cflags`	Specifies additional compiler flags that will be used when compiling C sources.
`cxxflags`	Specifies additional compiler flags that will be used when compiling C++ sources.
`compileflags`	Specifies additional compiler flags that will be used when compiling both C and C++ sources.
`linkflags`	Specifies additional command line options that will be passed to the linker.

Borland C++ Compiler

The `borland` module supports the command line C++ compiler included in C++ Builder 2006 product and earlier version of it, running on Microsoft Windows.

The supported products are listed below. The version reported by the command lines tools is also listed for reference.:

- C++ Builder 2006—5.8.2
- CBuilderX—5.6.5, 5.6.4 (depending on release)
- CBuilder6—5.6.4
- Free command line tools—5.5.1

The module is initialized using the following syntax:

```
using -borland -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the command is not specified, Boost.Build will search for a binary named **bcc32** in `PATH`.

The following options can be provided, using `<option-name>option-value` syntax:

`cflags`	Specifies additional compiler flags that will be used when compiling C sources.
`cxxflags`	Specifies additional compiler flags that will be used when compiling C++ sources.
`compileflags`	Specifies additional compiler flags that will be used when compiling both C and C++ sources.
`linkflags`	Specifies additional command line options that will be passed to the linker.

Comeau C/C++ Compiler

The `como-linux` and the `como-win` modules supports the Comeau C/C++ Compiler on Linux and Windows respectively.

The module is initialized using the following syntax:

```
using -como-linux -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the command is not specified, Boost.Build will search for a binary named **como** in `PATH`.

The following options can be provided, using `<option-name>option-value` syntax:

`cflags`	Specifies additional compiler flags that will be used when compiling C sources.
`cxxflags`	Specifies additional compiler flags that will be used when compiling C++ sources.
`compileflags`	Specifies additional compiler flags that will be used when compiling both C and C++ sources.
`linkflags`	Specifies additional command line options that will be passed to the linker.

Before using the Windows version of the compiler, you need to setup necessary environment variables per compiler's documentation. In particular, the `COMO_XXX_INCLUDE` variable should be set, where `XXX` corresponds to the used backend C compiler.

Code Warrior

The `cw` module support CodeWarrior compiler, originally produced by Metrowerks and presently developed by Freescale. Boost.Build supports only the versions of the compiler that target x86 processors. All such versions were released by Metrowerks before aquisition and are not sold any longer. The last version known to work is 9.4.

The module is initialized using the following syntax:

```
using -cw -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the command is not specified, Boost.Build will search for a binary named **mwcc** in default installation paths and in `PATH`.

The following options can be provided, using `<option-name>option-value` syntax:

`cflags`	Specifies additional compiler flags that will be used when compiling C sources.
`cxxflags`	Specifies additional compiler flags that will be used when compiling C++ sources.
`compileflags`	Specifies additional compiler flags that will be used when compiling both C and C++ sources.
`linkflags`	Specifies additional command line options that will be passed to the linker.
`root`	Specifies root directory of the compiler installation. This option is necessary only if it is not possible to detect this information from the compiler command—for example if the specified compiler command is a user script.
`setup`	The command that sets up environment variables prior to invoking the compiler. If not specified, **cwenv.bat** alongside the compiler binary will be used.
`compiler`	The command that compiles C and C++ sources. If not specified, **mwcc** will be used. The command will be invoked after the setup script was executed and adjusted the `PATH` variable.
`linker`	The command that links executables and dynamic libraries. If not specified, **mwld** will be used. The command will be invoked after the setup script was executed and adjusted the `PATH` variable.

Digital Mars C/C++ Compiler

The `dmc` module supports the Digital Mars C++ compiler.

The module is initialized using the following syntax:

```
using -dmc -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the command is not specified, Boost.Build will search for a binary named **dmc** in `PATH`.

The following options can be provided, using `<option-name>option-value` syntax:

`cflags`	Specifies additional compiler flags that will be used when compiling C sources.
`cxxflags`	Specifies additional compiler flags that will be used when compiling C++ sources.

`compileflags` Specifies additional compiler flags that will be used when compiling both C and C++ sources.

`linkflags` Specifies additional command line options that will be passed to the linker.

HP C++ Compiler for Tru64 Unix

The `hp_cxx` modules supports the HP C++ Compiler for Tru64 Unix.

The module is initialized using the following syntax:

```
using -hp_cxx -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the command is not specified, Boost.Build will search for a binary named **hp_cxx** in `PATH`.

The following options can be provided, using *<option-name>option-value* syntax:

`cflags` Specifies additional compiler flags that will be used when compiling C sources.

`cxxflags` Specifies additional compiler flags that will be used when compiling C++ sources.

`compileflags` Specifies additional compiler flags that will be used when compiling both C and C++ sources.

`linkflags` Specifies additional command line options that will be passed to the linker.

Sun Studio

The `sun` module supports the Sun Studio C++ compilers for the Solaris OS.

The module is initialized using the following syntax:

```
using -sun -: -[version] -: -[c++-compile-command] -: -[compiler -options] -;
```

This statement may be repeated several times, if you want to configure several versions of the compiler.

If the command is not specified, Boost.Build will search for a binary named **CC** in `/opt/SUNWspro/bin` and in `PATH`.

When using this compiler on complex C++ code, such as the Boost C++ library, it is recommended to specify the following options when intializing the `sun` module:

```
-library=stlport4 --features=tmplife --features=tmplrefstatic
 - - - - - - - - - -
```

See the Sun C++ Frontend Tales for details.

The following options can be provided, using *<option-name>option-value* syntax:

`cflags` Specifies additional compiler flags that will be used when compiling C sources.

`cxxflags` Specifies additional compiler flags that will be used when compiling C++ sources.

`compileflags` Specifies additional compiler flags that will be used when compiling both C and C++ sources.

`linkflags` Specifies additional command line options that will be passed to the linker.

Starting with Sun Studio 12, you can create 64-bit applications by using the `address-model=64` property.

IBM Visual Age

The `vacpp` module supports the IBM Visual Age C++ Compiler, for the AIX operating system. Versions 7.1 and 8.0 are known to work.

The module is initialized using the following syntax:

```
using -vacpp -;
```

The module does not accept any initialization options. The compiler should be installed in the `/usr/vacpp/bin` directory.

Later versions of Visual Age are known as XL C/C++. They were not tested with the the `vacpp` module.

Third-party libraries

Boost.Build provides special support for some third-party C++ libraries, documented below.

STLport library

The STLport library is an alternative implementation of C++ runtime library. Boost.Build supports using that library on Windows platfrom. Linux is hampered by different naming of libraries in each STLport version and is not officially supported.

Before using STLport, you need to configure it in `user-config.jam` using the following syntax:

```
using -stlport -: -[version] -: -header-path -: -[library-path] -;
```

Where *`version`* is the version of STLport, for example `5.1.4`, *`headers`* is the location where STLport headers can be found, and *`libraries`* is the location where STLport libraries can be found. The version should always be provided, and the library path should be provided if you're using STLport's implementation of iostreams. Note that STLport 5.* always uses its own iostream implementation, so the library path is required.

When STLport is configured, you can build with STLport by requesting `stdlib=stlport` on the command line.

Build process

The general overview of the build process was given in the user documentation. This section provides additional details, and some specific rules.

To recap, building a target with specific properties includes the following steps:

1. applying default build,
2. selecting the main target alternative to use,
3. determining "common" properties,
4. building targets referred by the sources list and dependency properties,
5. adding the usage requirements produces when building dependencies to the "common" properties,
6. building the target using generators,

7. computing the usage requirements to be returned.

Alternative selection

When there are several alternatives, one of them must be selected. The process is as follows:

1. For each alternative *condition* is defined as the set of base properties in requirements. [Note: it might be better to specify the condition explicitly, as in conditional requirements].
2. An alternative is viable only if all properties in condition are present in build request.
3. If there's one viable alternative, it's choosen. Otherwise, an attempt is made to find one best alternative. An alternative a is better than another alternative b, iff the set of properties in b's condition is a strict subset of the set of properities of 'a's condition. If there's one viable alternative, which is better than all others, it's selected. Otherwise, an error is reported.

Determining common properties

The "common" properties is a somewhat artificial term. Those are the intermediate property set from which both the build request for dependencies and properties for building the target are derived.

Since default build and alternatives are already handled, we have only two inputs: build requests and requirements. Here are the rules about common properties.

1. Non-free feature can have only one value
2. A non-conditional property in requirement in always present in common properties.
3. A property in build request is present in common properties, unless (2) tells otherwise.
4. If either build request, or requirements (non-conditional or conditional) include an expandable property (either composite, or property with specified subfeature value), the behaviour is equivalent to explicitly adding all expanded properties to build request or requirements.
5. If requirements include a conditional property, and condiiton of this property is true in context of common properties, then the conditional property should be in common properties as well.
6. If no value for a feature is given by other rules here, it has default value in common properties.

Those rules are declarative, they don't specify how to compute the common properties. However, they provide enough information for the user. The important point is the handling of conditional requirements. The condition can be satisfied either by property in build request, by non-conditional requirements, or even by another conditional property. For example, the following example works as expected:

```
exe -a -: -a.cpp
 - - - - - -: -<toolset>gcc:<variant>release
 - - - - - - - -<variant>release:<define>FOO -;
```

Definitions

Features and properties

A *feature* is a normalized (toolset-independent) aspect of a build configuration, such as whether inlining is enabled. Feature names may not contain the '>' character.

Each feature in a build configuration has one or more associated *value*s. Feature values for non-free features may not contain the '<', ':', or '=' characters. Feature values for free features may not contain the '<' character.

A *property* is a (feature,value) pair, expressed as <feature>value.

A *subfeature* is a feature that only exists in the presence of its parent feature, and whose identity can be derived (in the context of its parent) from its value. A subfeature's parent can never be another subfeature. Thus, features and their subfeatures form a two-level hierarchy.

A *value-string* for a feature **F** is a string of the form `value-subvalue1-subvalue2...-subvalueN`, where `value` is a legal value for **F** and `subvalue1...subvalueN` are legal values of some of **F**'s subfeatures. For example, the properties `<toolset>gcc <toolset-version>3.0.1` can be expressed more conscisely using a value-string, as `<toolset>gcc-3.0.1`.

A *property set* is a set of properties (i.e. a collection without duplicates), for instance: `<toolset>gcc <runtime-link>static`.

A *property path* is a property set whose elements have been joined into a single string separated by slashes. A property path representation of the previous example would be `<toolset>gcc/<runtime-link>static`.

A *build specification* is a property set that fully describes the set of features used to build a target.

Property Validity

For free features, all values are valid. For all other features, the valid values are explicitly specified, and the build system will report an error for the use of an invalid feature-value. Subproperty validity may be restricted so that certain values are valid only in the presence of certain other subproperties. For example, it is possible to specify that the `<gcc-target>mingw` property is only valid in the presence of `<gcc-version>2.95.2`.

Feature Attributes

Each feature has a collection of zero or more of the following attributes. Feature attributes are low-level descriptions of how the build system should interpret a feature's values when they appear in a build request. We also refer to the attributes of properties, so that an *incidental* property, for example, is one whose feature has the *incidental* attribute.

- *incidental*

  Incidental features are assumed not to affect build products at all. As a consequence, the build system may use the same file for targets whose build specification differs only in incidental features. A feature that controls a compiler's warning level is one example of a likely incidental feature.

  Non-incidental features are assumed to affect build products, so the files for targets whose build specification differs in non-incidental features are placed in different directories as described in "target paths" below. [where?]

- *propagated*

  Features of this kind are propagated to dependencies. That is, if a main target is built using a propagated property, the build systems attempts to use the same property when building any of its dependencies as part of that main target. For instance, when an optimized exectuable is requested, one usually wants it to be linked with optimized libraries. Thus, the `<optimization>` feature is propagated.

- *free*

  Most features have a finite set of allowed values, and can only take on a single value from that set in a given build specification. Free features, on the other hand, can have several values at a time and each value can be an arbitrary string. For example, it is possible to have several preprocessor symbols defined simultaneously:

```
<define>NDEBUG=1 -<define>HAS_CONFIG_H=1
```

- *optional*

An optional feature is a feature that is not required to appear in a build specification. Every non-optional non-free feature has a default value that is used when a value for the feature is not otherwise specified, either in a target's requirements or in the user's build request. [A feature's default value is given by the first value listed in the feature's declaration. -- move this elsewhere - dwa]

- *symmetric*

  A symmetric feature's default value is not automatically included in build variants. Normally a feature only generates a subvariant directory when its value differs from the value specified by the build variant, leading to an assymmetric subvariant directory structure for certain values of the feature. A symmetric feature, when relevant to the toolset, always generates a corresponding subvariant directory.

- *path*

  The value of a path feature specifies a path. The path is treated as relative to the directory of Jamfile where path feature is used and is translated appropriately by the build system when the build is invoked from a different directory

- *implicit*

  Values of implicit features alone identify the feature. For example, a user is not required to write "<toolset>gcc", but can simply write "gcc". Implicit feature names also don't appear in variant paths, although the values do. Thus: bin/gcc/... as opposed to bin/toolset-gcc/.... There should typically be only a few such features, to avoid possible name clashes.

- *composite*

  Composite features actually correspond to groups of properties. For example, a build variant is a composite feature. When generating targets from a set of build properties, composite features are recursively expanded and *added* to the build property set, so rules can find them if necessary. Non-composite non-free features override components of composite features in a build property set.

- *dependency*

  The value of dependency feature if a target reference. When used for building of a main target, the value of dependency feature is treated as additional dependency.

  For example, dependency features allow to state that library A depends on library B. As the result, whenever an application will link to A, it will also link to B. Specifying B as dependency of A is different from adding B to the sources of A.

Features that are neither free nor incidental are called *base* features.

Feature Declaration

The low-level feature declaration interface is the `feature` rule from the `feature` module:

```
rule -feature -( -name -: -allowed-values -* -: -attributes -* -)
```

A feature's allowed-values may be extended with the `feature.extend` rule.

Build Variants

A build variant, or (simply variant) is a special kind of composite feature that automatically incorporates the default values of features that . Typically you'll want at least two separate variants: one for debugging, and one for your release code. [Volodya says: "Yea, we'd need to mention that it's a composite feature and describe how they are declared, in pacticular that default values of non-optional features are incorporated into build variant automagically. Also, do we wan't some variant inheritance/extension/templates. I don't remember how it works in V1, so can't document this for V2.". Will clean up soon -DWA]

Property refinement

When a target with certain properties is requested, and that target requires some set of properties, it is needed to find the set of properties to use for building. This process is called *property refinement* and is performed by these rules

1. Each property in the required set is added to the original property set
2. If the original property set includes property with a different value of non free feature, that property is removed.

Conditional properties

Sometime it's desirable to apply certain requirements only for a specific combination of other properties. For example, one of compilers that you use issues a pointless warning that you want to suppress by passing a command line option to it. You would not want to pass that option to other compilers. Conditional properties allow you to do just that. Their syntax is:

```
 - - - - - - - -property -( -"," -property -) -* -":" -property
 - - - - - -
```

For example, the problem above would be solved by:

```
exe -hello -: -hello.cpp -: -<toolset>yfc:<cxxflags>-disable-pointless-warning -;
```

The syntax also allows several properties in the condition, for example:

```
exe -hello -: -hello.cpp -: -<os>NT,<toolset>gcc:<link>static -;
```

Target identifiers and references

Target identifier is used to denote a target. The syntax is:

```
target-id --> -(project-id -| -target-name -| -file-name -)
  - - - - - - - - - - - - - -| -(project-id -| -directory-name) -"//" -target-name
project-id --> -path
target-name --> -path
file-name --> -path
directory-name --> -path
```

This grammar allows some elements to be recognized as either

- project id (at this point, all project ids start with slash).
- name of target declared in current Jamfile (note that target names may include slash).
- a regular file, denoted by absolute name or name relative to project's sources location.

To determine the real meaning a check is made if project-id by the specified name exists, and then if main target of that name exists. For example, valid target ids might be:

```
a - - - - - - - - - - - - -
```

```
/home/ghost/build/lr_library//parser --- -target -in -specific -project
```

Rationale: Target is separated from project by special separator (not just slash), because:

- It emphasises that projects and targets are different things.
- It allows to have main target names with slashes.

Target reference is used to specify a source target, and may additionally specify desired properties for that target. It has this syntax:

```
target-reference --> -target-id -[ -"/" -requested-properties -]
requested-properties --> -property-path
```

For example,

```
 - - - - - - - - - -exe -compiler -: -compiler.cpp -libs/cmdline/<optimization>space -;
 - - - - - - - - -
```

would cause the version of `cmdline` library, optimized for space, to be linked in even if the `compiler` executable is build with optimization for speed.

Extender Manual

Introduction

This section explains how to extend Boost.Build to accomodate your local requirements—primarily to add support for non-standard tools you have. Before we start, be sure you have read and understoon the concept of metatarget, the section called "Concepts", which is critical to understanding the remaining material.

The current version of Boost.Build has three levels of targets, listed below.

metatarget	Object that is created from declarations in Jamfiles. May be called with a set of properties to produce concrete targets.
concrete target	Object that corresponds to a file or an action.
jam target	Low-level concrete target that is specific to Boost.Jam build engine. Essentially a string—most often a name of file.

In most cases, you will only have to deal with concrete targets and the process that creates concrete targets from metatargets. Extending metatarget level is rarely required. The jam targets are typically only used inside the command line patterns.

Warning

All of the Boost.Jam target-related builtin functions, like `DEPENDS` or `ALWAYS` operate on jam targets. Applying them to metatargets or concrete targets has no effect.

Metatargets

Metatarget is an object that records information specified in Jamfile, such as metatarget kind, name, sources and properties, and can be called with specific properties to generate concrete targets. At the code level it is represented by an instance of class derived from `abstract-target`. [1]

The `generate` method takes the build properties (as an instance of the `property-set` class) and returns a list containing:

- As front element—Usage-requirements from this invocation (an instance of `property-set`)
- As subsequent elements—created concrete targets (instances of the `virtual-target` class.)

It's possible to lookup a metataget by target-id using the `targets.resolve-reference` function, and the `targets.generate-from-reference` function can both lookup and generate a metatarget.

The `abstract-target` class has three immediate derived classes:

- `project-target` that corresponds to a project and is not intended for further subclassing. The `generate` method of this class builds all targets in the project that are not marked as explicit.
- `main-target` corresponds to a target in a project and contains one or more target alternatives. This class also should not be subclassed. The `generate` method of this class selects an alternative to build, and calls the `generate` method of that alternative.
- `basic-target` corresponds to a specific target alternative. This is base class, with a number of derived classes. The `gener-`

[1] This name is historic, and will be eventuall changed to `metatarget`

ate method processes the target requirements and requested build properties to determine final properties for the target, builds all sources, and finally calls the abstract construct method with the list of source virtual targets, and the final properties.

The instances of the project-target and main-target classes are created implicitly—when loading a new Jamfiles, or when a new target alternative with as-yet unknown name is created. The instances of the classes derived from basic-target are typically created when Jamfile calls a *metatarget rule*, such as such as exe.

It it permissible to create a custom class derived from basic-target and create new metatarget rule that creates instance of such target. However, in the majority of cases, a specific subclass of basic-target— typed-target is used. That class is associated with a *type* and relays to *generators* to construct concrete targets of that type. This process will be explained below. When a new type is declared, a new metatarget rule is automatically defined. That rule creates new instance of type-target, associated with that type.

Concrete targets

Concrete targets are represented by instance of classes derived from virtual-target. The most commonly used subclass is file-target. A file target is associated with an action that creates it— an instance of the action class. The action, in turn, hold a list of source targets. It also holds the property-set instance with the build properties that should be used for the action.

Here's an example of creating a target from another target, source

```
local -a -= -[ -new -action -$(source) -: -common.copy -: -$(property-set) -] -;
local -t -= -[ -new -file-target -$(name) -: -CPP -: -$(project) -: -$(a) -] -;
```

The first line creates an instance of the action> class. The first parameter is the list of sources. The second parameter is the name a jam-level action. The third parameter is the property-set applying to this action. The second line creates a target. We specifie a name, a type and a project. We also pass the action object created earlier. If the action creates several targets, we can repeat the second line several times.

In some cases, code that creates concrete targets may be invoked more than once with the same properties. Returning to different instance of file-target that correspond to the same file clearly will result in problems. Therefore, whenever returning targets you should pass them via the virtual-target.register function, that will replace targets with previously created identical ones, as necessary.[2] Here are a couple of examples:

```
return -[ -virtual-target.register -$(t) -] -;
return -[ -sequence.transform -virtual-target.register -: -$(targets) -] -;
```

Generators

In theory, every kind of metatarget in Boost.Build (like exe, lib or obj) could be implemented by writing a new metatarget class that, independently of the other code, figures what files to produce and what commands to use. However, that would be rather inflexible. For example, adding support for a new compiler would require editing several metatargets.

In practice, most files have specific types, and most tools consume and produce files of specific type. To take advantage of this fact, Boost.Build defines concept of target type and *generators*, and has special metatarget class typed-target. Target type is merely an identifier. It is associated with a set of file extensions that correspond to that type. Generator is an abstraction of a tool. It advertises the types it produces and, if called with a set of input target, tries to construct output targets of the advertised types. Finally, typed-target is associated with specific target type, and relays the generator (or generators) for that type.

A generator is an instance of a class derived from generator. The generator class itself is suitable for common cases. You can define derived classes for custom scenarios.

[2]This create-then-register pattern is caused by limitations of the Boost.Jam language. Python port is likely to never create duplicate targets.

Example: 1-to-1 generator

Say you're writing an application that generates C++ code. If you ever did this, you know that it's not nice. Embedding large portions of C++ code in string literals is very awkward. A much better solution is:

1. Write the template of the code to be generated, leaving placeholders at the points that will change
2. Access the template in your application and replace placeholders with appropriate text.
3. Write the result.

It's quite easy to achieve. You write special verbatim files that are just C++, except that the very first line of the file contains the name of a variable that should be generated. A simple tool is created that takes a verbatim file and creates a cpp file with a single `char*` variable whose name is taken from the first line of the verbatim file and whose value is the file's properly quoted content.

Let's see what Boost.Build can do.

First off, Boost.Build has no idea about "verbatim files". So, you must register a new target type. The following code does it:

```
import -type -;
type.register -VERBATIM -: -verbatim -;
```

The first parameter to `type.register` gives the name of the declared type. By convention, it's uppercase. The second parameter is the suffix for files of this type. So, if Boost.Build sees `code.verbatim` in a list of sources, it knows that it's of type `VERBATIM`.

Next, you tell Boost.Build that the verbatim files can be transformed into C++ files in one build step. A *generator* is a template for a build step that transforms targets of one type (or set of types) into another. Our generator will be called `verbatim.inline-file`; it transforms `VERBATIM` files into `CPP` files:

```
import -generators -;
generators.register-standard -verbatim.inline-file -: -VERBATIM -: -CPP -;
```

Lastly, you have to inform Boost.Build about the shell commands used to make that transformation. That's done with an `actions` declaration.

```
actions -inline-file
{
 - - - -"./inline-file.py" -$(<) -$(>)
}
```

Now, we're ready to tie it all together. Put all the code above in file `verbatim.jam`, add `import verbatim ;` to `Jamroot.jam`, and it's possible to write the following in your Jamfile:

```
exe -codegen -: -codegen.cpp -class_template.verbatim -usage.verbatim -;
```

The listed verbatim files will be automatically converted into C++ source files, compiled and then linked to the codegen executable.

In subsequent sections, we will extend this example, and review all the mechanisms in detail. The complete code is available in the `example/customization` directory.

Target types

The first thing we did in the intruduction was declaring a new target type:

```
import -type -;
type.register -VERBATIM -: -verbatim -;
```

The type is the most important property of a target. Boost.Build can automatically generate necessary build actions only because you specify the desired type (using the different main target rules), and because Boost.Build can guess the type of sources from their extensions.

The first two parameters for the `type.register` rule are the name of new type and the list of extensions associated with it. A file with an extension from the list will have the given target type. In the case where a target of the declared type is generated from other sources, the first specified extension will be used.

Sometimes you want to change the suffix used for generated targets depending on build properties, such as toolset. For example, some compiler uses extension `elf` for executable files. You can use the `type.set-generated-target-suffix` rule:

```
type.set-generated-target-suffix -EXE -: -<toolset>elf -: -elf -;
```

A new target type can be inherited from an existing one.

```
type.register -PLUGIN -: -: -SHARED_LIB -;
```

The above code defines a new type derived from `SHARED_LIB`. Initially, the new type inherits all the properties of the base type - in particular generators and suffix. Typically, you'll change the new type in some way. For example, using `type.set-generated-target-suffix` you can set the suffix for the new type. Or you can write special a generator for the new type. For example, it can generate additional metainformation for the plugin. In either way, the `PLUGIN` type can be used whenever `SHARED_LIB` can. For example, you can directly link plugins to an application.

A type can be defined as "main", in which case Boost.Build will automatically declare a main target rule for building targets of that type. More details can be found later.

Scanners

Sometimes, a file can refer to other files via some include system. To make Boost.Build track dependencies between included files, you need to provide a scanner. The primary limitation is that only one scanner can be assigned to a target type.

First, we need to declare a new class for the scanner:

```
class -verbatim-scanner -: -common-scanner
{
 - - - -rule -pattern -( -)
 - - - -{
 - - - - - - - -return -"//###include[ -]*\"([^\"]*)\"" -;
 - - - -}
}
```

All the complex logic is in the `common-scanner` class, and you only need to override the method that returns the regular expression to be used for scanning. The parentheses in the regular expression indicate which part of the string is the name of the included file. Only the first parenthesized group in the regular expression will be recognized; if you can't express everything you want that way, you can return multiple regular expressions, each of which contains a parenthesized group to be matched.

After that, we need to register our scanner class:

```
scanner.register -verbatim-scanner -: -include -;
```

```
```

The value of the second parameter, in this case `include`, specifies the properties that contain the list of paths that should be searched for the included files.

Finally, we assign the new scanner to the `VERBATIM` target type:

```
type.set-scanner -VERBATIM -: -verbatim-scanner -;
```

That's enough for scanning include dependencies.

Tools and generators

This section will describe how Boost.Build can be extended to support new tools.

For each additional tool, a Boost.Build object called generator must be created. That object has specific types of targets that it accepts and produces. Using that information, Boost.Build is able to automatically invoke the generator. For example, if you declare a generator that takes a target of the type `D` and produces a target of the type `OBJ`, when placing a file with extention `.d` in a list of sources will cause Boost.Build to invoke your generator, and then to link the resulting object file into an application. (Of course, this requires that you specify that the `.d` extension corresponds to the `D` type.)

Each generator should be an instance of a class derived from the `generator` class. In the simplest case, you don't need to create a derived class, but simply create an instance of the `generator` class. Let's review the example we've seen in the introduction.

```
import -generators -;
generators.register-standard -verbatim.inline-file -: -VERBATIM -: -CPP -;
actions -inline-file
{
 - - - -"./inline-file.py" -$(<) -$(>)
}
```

We declare a standard generator, specifying its id, the source type and the target type. When invoked, the generator will create a target of type `CPP` with a source target of type `VERBATIM` as the only source. But what command will be used to actually generate the file? In bjam, actions are specified using named "actions" blocks and the name of the action block should be specified when creating targets. By convention, generators use the same name of the action block as their own id. So, in above example, the "inline-file" actions block will be used to convert the source into the target.

There are two primary kinds of generators: standard and composing, which are registered with the `generators.register-standard` and the `generators.register-composing` rules, respectively. For example:

```
generators.register-standard -verbatim.inline-file -: -VERBATIM -: -CPP -;
generators.register-composing -mex.mex -: -CPP -LIB -: -MEX -;
```

The first (standard) generator takes a *single* source of type `VERBATIM` and produces a result. The second (composing) generator takes any number of sources, which can have either the `CPP` or the `LIB` type. Composing generators are typically used for generating top-level target type. For example, the first generator invoked when building an `exe` target is a composing generator corresponding to the proper linker.

You should also know about two specific functions for registering generators: `generators.register-c-compiler` and `generators.register-linker`. The first sets up header dependecy scanning for C files, and the seconds handles various complexities like searched libraries. For that reason, you should always use those functions when adding support for compilers and linkers.

(Need a note about UNIX)

Custom generator classes

The standard generators allows you to specify source and target types, an action, and a set of flags. If you need anything more

complex, you need to create a new generator class with your own logic. Then, you have to create an instance of that class and register it. Here's an example how you can create your own generator class:

```
class -custom-generator -: -generator
{
 - - - -rule -__init__ -( -* -: -* -)
 - - - -{
 - - - - - - - -generator.__init__ -$(1) -: -$(2) -: -$(3) -: -$(4) -: -$(5) -: -$(6) -: -$(7)
 -: -$(8) -: -$(9) -;
 - - - -}

}

generators.register
 - -[ -new -custom-generator -verbatim.inline-file -: -VERBATIM -: -CPP -] -;
```

This generator will work exactly like the `verbatim.inline-file` generator we've defined above, but it's possible to customize the behaviour by overriding methods of the `generator` class.

There are two methods of interest. The `run` method is responsible for the overall process - it takes a number of source targets, converts them to the right types, and creates the result. The `generated-targets` method is called when all sources are converted to the right types to actually create the result.

The `generated-targets` method can be overridden when you want to add additional properties to the generated targets or use additional sources. For a real-life example, suppose you have a program analysis tool that should be given a name of executable and the list of all sources. Naturally, you don't want to list all source files manually. Here's how the `generated-targets` method can find the list of sources automatically:

```
class -itrace-generator -: -generator -{
....
 - - - -rule -generated-targets -( -sources -+ -: -property-set -: -project -name -? -)
 - - - -{
 - - - - - - - -local -leaves -;
 - - - - - - - -local -temp -= -[ -virtual-target.traverse -$(sources[1]) -: -: -include-sourc
es -] -;
 - - - - - - - -for -local -t -in -$(temp)
 - - - - - - - -{
 - - - - - - - - - - - -if -! -[ -$(t).action -]
 - - - - - - - - - - - -{
 - - - - - - - - - - - - - - -leaves -+= -$(t) -;
 - - - - - - - - - - - -}
 - - - - - - -}
 - - - - - - -return -[ -generator.generated-targets -$(sources) -$(leafs)
 - - - - - - - - - -: -$(property-set) -: -$(project) -$(name) -] -;
 - - - -}
}
generators.register -[ -new -itrace-generator -nm.itrace -: -EXE -: -ITRACE -] -;
```

The `generated-targets` method will be called with a single source target of type `EXE`. The call to `virtual-target.traverse` will return all targets the executable depends on, and we further find files that are not produced from anything. The found targets are added to the sources.

The `run` method can be overriden to completely customize the way the generator works. In particular, the conversion of sources to the desired types can be completely customized. Here's another real example. Tests for the Boost Python library usually consist of two parts: a Python program and a C++ file. The C++ file is compiled to Python extension that is loaded by the Python program. But in the likely case that both files have the same name, the created Python extension must be renamed. Otherwise, the Python program will import itself, not the extension. Here's how it can be done:

```
rule -run -( -project -name -? -: -property-set -: -sources -* -)
{
 - - - -local -python -;
 - - - -for -local -s -in -$(sources)
 - - - -{
 - - - - - - - -if -[ -$(s).type -] -= -PY
 - - - - - - - -{
 - - - - - - - - - - - -python -= -$(s) -;
```

```
 - - - - - - - - -}
 - - - -}
 - - - -
 - - - -local -libs -;
 - - - -for -local -s -in -$(sources)
 - - - -{
 - - - - - - - - -if -[ -type.is-derived -[ -$(s).type -] -LIB -]
 - - - - - - - - -{
 - - - - - - - - - - - - -libs -+= -$(s) -;
 - - - - - - - - -}
 - - - -}

 - - - -local -new-sources -;
 - - - -for -local -s -in -$(sources)
 - - - -{
 - - - - - - - - -if -[ -type.is-derived -[ -$(s).type -] -CPP -]
 - - - - - - - - -{
 - - - - - - - - - - - - -local -name -= -[ -$(s).name -] -; - - - -# -get -the -target's -basen
ame
 - - - - - - - - - - - - -if -$(name) -= -[ -$(python).name -]
 - - - - - - - - - - - - -{
 - - - - - - - - - - - - - - - -name -= -$(name)_ext -; - - - - - - - -# -rename -the -target
 - - - - - - - - - - - - -}
 - - - - - - - - - - - - -new-sources -+= -[ -generators.construct -$(project) -$(name) -:
 - - - - - - - - - - - - - - - -PYTHON_EXTENSION -: -$(property-set) -: -$(s) -$(libs) -] -;
 - - - - - - - - -}
 - - - -}

 - - - -result -= -[ -construct-result -$(python) -$(new-sources) -: -$(project) -$(name)
 - - - - - - - - - - - - - - - - - - - -: -$(property-set) -] -;
}
```

First, we separate all source into python files, libraries and C++ sources. For each C++ source we create a separate Python extension by calling `generators.construct` and passing the C++ source and the libraries. At this point, we also change the extension's name, if necessary.

Features

Often, we need to control the options passed the invoked tools. This is done with features. Consider an example:

```
# -Declare -a -new -free -feature
import -feature -: -feature -;
feature -verbatim-options -: -: -free -;

# -Cause -the -value -of -the -'verbatim-options' -feature -to -be
# -available -as -'OPTIONS' -variable -inside -verbatim.inline-file
import -toolset -: -flags -;
flags -verbatim.inline-file -OPTIONS -<verbatim-options> -;

# -Use -the -"OPTIONS" -variable
actions -inline-file
{
 - - - -"./inline-file.py" -$(OPTIONS) -$(<) -$(>)
}
```

We first define a new feature. Then, the `flags` invocation says that whenever verbatin.inline-file action is run, the value of the `verbatim-options` feature will be added to the `OPTIONS` variable, and can be used inside the action body. You'd need to consult online help (--help) to find all the features of the `toolset.flags` rule.

Although you can define any set of features and interpret their values in any way, Boost.Build suggests the following coding standard for designing features.

Most features should have a fixed set of values that is portable (tool neutral) across the class of tools they are designed to work with. The user does not have to adjust the values for a exact tool. For example, `<optimization>speed` has the same meaning for all C++ compilers and the user does not have to worry about the exact options passed to the compiler's command line.

Besides such portable features there are special 'raw' features that allow the user to pass any value to the command line parameters

for a particular tool, if so desired. For example, the `<cxxflags>` feature allows you to pass any command line options to a C++ compiler. The `<include>` feature allows you to pass any string preceded by `-I` and the interpretation is tool-specific. (See the section called “ Can I get capture external program output using a Boost.Jam variable? ” for an example of very smart usage of that feature). Of course one should always strive to use portable features, but these are still be provided as a backdoor just to make sure Boost.Build does not take away any control from the user.

Using portable features is a good idea because:

- When a portable feature is given a fixed set of values, you can build your project with two different settings of the feature and Boost.Build will automatically use two different directories for generated files. Boost.Build does not try to separate targets built with different raw options.
- Unlike with “raw” features, you don't need to use specific command-line flags in your Jamfile, and it will be more likely to work with other tools.

Steps for adding a feauture

Adding a feature requires three steps:

1. Declaring a feature. For that, the "feature.feature" rule is used. You have to decide on the set of feature attributes:
   - if you want a feature value set for one target to automaticaly propagate to its dependant targets then make it “propagated”.
   - if a feature does not have a fixed list of values, it must be “free.” For example, the `include` feature is a free feature.
   - if a feature is used to refer to a path relative to the Jamfile, it must be a “path” feature. Such features will also get their values automatically converted to Boost Build's internal path representation. For example, `include` is a path feature.
   - if feature is used to refer to some target, it must be a “dependency” feature.
2. Representing the feature value in a target-specific variable. Build actions are command templates modified by Boost.Jam variable expansions. The `toolset.flags` rule sets a target-specific variable to the value of a feature.
3. Using the variable. The variable set in step 2 can be used in a build action to form command parameters or files.

Another example

Here's another example. Let's see how we can make a feature that refers to a target. For example, when linking dynamic libraries on Windows, one sometimes needs to specify a "DEF file", telling what functions should be exported. It would be nice to use this file like this:

```
- - - - - - - -lib -a -: -a.cpp -: -<def-file>a.def -;
```

Actually, this feature is already supported, but anyway...

1. Since the feature refers to a target, it must be "dependency".

```
feature -def-file -: -: -free -dependency -;
```

2. One of the toolsets that cares about DEF files is msvc. The following line should be added to it.

```
flags -msvc.link -DEF_FILE -<def-file> -;
```

```
```

3. Since the DEF_FILE variable is not used by the msvc.link action, we need to modify it to be:

```
actions -link -bind -DEF_FILE
{
 - - - -$(.LD) -.... -/DEF:$(DEF_FILE) -....
}
```

Note the `bind DEF_FILE` part. It tells bjam to translate the internal target name in `DEF_FILE` to a corresponding filename in the `link` action. Without it the expansion of `$(DEF_FILE)` would be a strange symbol that is not likely to make sense for the linker.

We are almost done, but we should stop for a small workaround. Add the following code to msvc.jam

```
rule -link
{
 - - - -DEPENDS -$(<) -: -[ -on -$(<) -return -$(DEF_FILE) -] -;
}
```

This is needed to accomodate some bug in bjam, which hopefully will be fixed one day.

Variants and composite features.

Sometimes you want to create a shortcut for some set of features. For example, `release` is a value of `<variant>` and is a shortcut for a set of features.

It is possible to define your own build variants. For example:

```
variant -crazy -: -<optimization>speed -<inlining>off
 - - - - - - - - - - - - - - - -<debug-symbols>on -<profiling>on -;
```

will define a new variant with the specified set of properties. You can also extend an existing variant:

```
variant -super_release -: -release -: -<define>USE_ASM -;
```

In this case, `super_release` will expand to all properties specified by `release`, and the additional one you've specified.

You are not restricted to using the `variant` feature only. Here's example that defines a brand new feature:

```
feature -parallelism -: -mpi -fake -none -: -composite -link-incompatible -;
feature.compose -<parallelism>mpi -: -<library>/mpi//mpi/<parallelism>none -;
feature.compose -<parallelism>fake -: -<library>/mpi//fake/<parallelism>none -;
```

This will allow you to specify the value of feature `parallelism`, which will expand to link to the necessary library.

Main target rules

A main target rule (e.g “`exe`” Or “`lib`”) creates a top-level target. It's quite likely that you'll want to declare your own and there are two ways to do that.

The first way applies when your target rule should just produce a target of specific type. In that case, a rule is already defined for

you! When you define a new type, Boost.Build automatically defines a corresponding rule. The name of the rule is obtained from the name of the type, by downcasing all letters and replacing underscores with dashes. For example, if you create a module `obfuscate.jam` containing:

```
import -type -;
type.register -OBFUSCATED_CPP - -: -ocpp -;

import -generators -;
generators.register-standard -obfuscate.file -: -CPP -: -OBFUSCATED_CPP -;
```

and import that module, you'll be able to use the rule "obfuscated-cpp" in Jamfiles, which will convert source to the OBFUSCATED_CPP type.

The second way is to write a wrapper rule that calls any of the existing rules. For example, suppose you have only one library per directory and want all cpp files in the directory to be compiled into that library. You can achieve this effect using:

```
lib -codegen -: -[ -glob -*.cpp -] -;
```

If you want to make it even simpler, you could add the following definition to the `Jamroot.jam` file:

```
rule -glib -( -name -: -extra-sources -* -: -requirements -* -)
{
 - - - -lib -$(name) -: -[ -glob -*.cpp -] -$(extra-sources) -: -$(requirements) -;
}
```

allowing you to reduce the Jamfile to just

```
glib -codegen -;
```

Note that because you can associate a custom generator with a target type, the logic of building can be rather complicated. For example, the `boostbook` module declares a target type `BOOSTBOOK_MAIN` and a custom generator for that type. You can use that as example if your main target rule is non-trivial.

Toolset modules

If your extensions will be used only on one project, they can be placed in a separate `.jam` file and imported by your `Jamroot.jam`. If the extensions will be used on many projects, users will thank you for a finishing touch.

The `using` rule provides a standard mechanism for loading and configuring extensions. To make it work, your module should provide an `init` rule. The rule will be called with the same parameters that were passed to the `using` rule. The set of allowed parameters is determined by you. For example, you can allow the user to specify paths, tool versions, and other options.

Here are some guidelines that help to make Boost.Build more consistent:

- The `init` rule should never fail. Even if the user provided an incorrect path, you should emit a warning and go on. Configuration may be shared between different machines, and wrong values on one machine can be OK on another.
- Prefer specifying the command to be executed to specifying the tool's installation path. First of all, this gives more control: it's possible to specify

```
/usr/bin/g++-snapshot
time -g++
```

as the command. Second, while some tools have a logical "installation root", it's better if the user doesn't have to remember whether a specific tool requires a full command or a path.

- Check for multiple initialization. A user can try to initialize the module several times. You need to check for this and decide what to do. Typically, unless you support several versions of a tool, duplicate initialization is a user error. If the tool's version can be specified during initialization, make sure the version is either always specified, or never specified (in which case the tool is initialied only once). For example, if you allow:

```
using -yfc -;
using -yfc -: -3.3 -;
using -yfc -: -3.4 -;
```

  Then it's not clear if the first initialization corresponds to version 3.3 of the tool, version 3.4 of the tool, or some other version. This can lead to building twice with the same version.

- If possible, `init` must be callable with no parameters. In which case, it should try to autodetect all the necessary information, for example, by looking for a tool in `PATH` or in common installation locations. Often this is possible and allows the user to simply write:

```
using -yfc -;
```

- Consider using facilities in the `tools/common` module. You can take a look at how `tools/gcc.jam` uses that module in the `init` rule.

Frequently Asked Questions

How do I get the current value of feature in Jamfile?

This is not possible, since Jamfile does not have "current" value of any feature, be it toolset, build variant or anything else. For a single invocation of `bjam`, any given main target can be built with several property sets. For example, user can request two build variants on the command line. Or one library is built as shared when used from one application, and as static when used from another. Each Jamfile is read only once so generally there is no single value of a feature you can access in Jamfile.

A feature has a specific value only when building a target, and there are two ways you can use that value:

- Use conditional requirements or indirect conditional requirements. See the section called “Requirements” [21].
- Define a custom generator and a custom main target type. The custom generator can do arbitrary processing or properties. See the *Extender Manual*.

I am getting a "Duplicate name of actual target" error. What does that mean?

The most likely case is that you are trying to compile the same file twice, with almost the same, but differing properties. For example:

```
exe -a -: -a.cpp -: -<include>/usr/local/include -;
exe -b -: -a.cpp -;
```

The above snippet requires two different compilations of `a.cpp`, which differ only in their `include` property. Since the `include` feature is declared as `free` Boost.Build does not create a separate build directory for each of its values and those two builds would both produce object files generated in the same build directory. Ignoring this and compiling the file only once would be dangerous as different includes could potentially cause completely different code to be compiled.

To solve this issue, you need to decide if the file should be compiled once or twice.

1. To compile the file only once, make sure that properties are the same for both target requests:

```
exe -a -: -a.cpp -: -<include>/usr/local/include -;
exe -b -: -a.cpp -: -<include>/usr/local/include -;
```

   or:

```
alias -a-with-include -: -a.cpp -: -<include>/usr/local/include -;
exe -a -: -a-with-include -;
exe -b -: -a-with-include -;
```

   or if you want the `includes` property not to affect how any other sources added for the built `a` and `b` executables would be compiled:

```
obj -a-obj -: -a.cpp -: -<include>/usr/local/include -;
exe -a -: -a-obj -;
exe -b -: -a-obj -;
```

Note that in both of these cases the `include` property will be applied only for building these object files and not any other sources that might be added for targets `a` and `b`.

2. To compile the file twice, you can tell Boost.Build to compile it to two separate object files like so:

```
- - - - - -obj -a_obj -: -a.cpp -: -<include>/usr/local/include -;
- - - - - -obj -b_obj -: -a.cpp -;
- - - - - -exe -a -: -a_obj -;
- - - - - -exe -b -: -b_obj -;
```

or you can make the object file targets local to the main target:

```
- - - - - -exe -a -: -[ -obj -a_obj -: -a.cpp -: -<include>/usr/local/include -] -;
- - - - - -exe -b -: -[ -obj -a_obj -: -a.cpp -] -;
```

which will cause Boost.Build to actually change the generated object file names a bit for you and thus avoid any conflicts.

Note that in both of these cases the `include` property will be applied only for building these object files and not any other sources that might be added for targets `a` and `b`.

A good question is why Boost.Build can not use some of the above approaches automatically. The problem is that such magic would only help in half of the cases, while in the other half it would be silently doing the wrong thing. It is simpler and safer to ask the user to clarify his intention in such cases.

Accessing environment variables

Many users would like to use environment variables in Jamfiles, for example, to control the location of external libraries. In many cases it is better to declare those external libraries in the site-config.jam file, as documented in the recipes section. However, if the users already have the environment variables set up, it may not be convenient for them to set up their site-config.jam files as well and using the environment variables might be reasonable.

Boost.Jam automatically imports all environment variables into its built-in .ENVIRON module so user can read them from there directly or by using the helper os.environ rule. For example:

```
import -os -;
local -unga-unga -= -[ -os.environ -UNGA_UNGA -] -;
ECHO -$(unga-unga) -;
```

or a bit more realistic:

```
import -os -;
local -SOME_LIBRARY_PATH -= -[ -os.environ -SOME_LIBRARY_PATH -] -;
exe -a -: -a.cpp -: -<include>$(SOME_LIBRARY_PATH) -;
```

How to control properties order?

For internal reasons, Boost.Build sorts all the properties alphabetically. This means that if you write:

```
exe -a -: -a.cpp -: -<include>b -<include>a -;
```

then the command line with first mention the `a` include directory, and then `b`, even though they are specified in the opposite order. In most cases, the user does not care. But sometimes the order of includes, or other properties, is important. For such cases, a spe-

cial syntax is provided:

```
exe -a -: -a.cpp -: -<include>a&&b -;
```

The && symbols separate property values and specify that their order should be preserved. You are advised to use this feature only when the order of properties really matters and not as a convenient shortcut. Using it everywhere might negatively affect performance.

How to control the library linking order on Unix?

On Unix-like operating systems, the order in which static libraries are specified when invoking the linker is important, because by default, the linker uses one pass though the libraries list. Passing the libraries in the incorrect order will lead to a link error. Further, this behaviour is often used to make one library override symbols from another. So, sometimes it is necessary to force specific library linking order.

Boost.Build tries to automatically compute the right order. The primary rule is that if library a "uses" library b, then library a will appear on the command line before library b. Library a is considered to use b if b is present either in the a library's sources or its usage is listed in its requirements. To explicitly specify the use relationship one can use the <use> feature. For example, both of the following lines will cause a to appear before b on the command line:

```
lib -a -: -a.cpp -b -;
lib -a -: -a.cpp -: -<use>b -;
```

The same approach works for searched libraries as well:

```
lib -z -;
lib -png -: -: -<use>z -;
exe -viewer -: -viewer -png -z -;
```

Can I get capture external program output using a Boost.Jam variable?

The SHELL builtin rule may be used for this purpose:

```
local -gtk_includes -= -[ -SHELL -"gtk-config ---cflags" -] -;
```

How to get the project root (a.k.a. Jamroot) location?

You might want to use your project's root location in your Jamfiles. To access it just declare a path constant in your Jamroot.jam file using:

```
path-constant -TOP -: -. -;
```

After that, the TOP variable can be used in every Jamfile.

How to change compilation flags for one file?

If one file must be compiled with special options, you need to explicitly declare an obj target for that file and then use that target

in your `exe` or `lib` target:

```
exe -a -: -a.cpp -b -;
obj -b -: -b.cpp -: -<optimization>off -;
```

Of course you can use other properties, for example to specify specific C/C++ compiler options:

```
exe -a -: -a.cpp -b -;
obj -b -: -b.cpp -: -<cflags>-g -;
```

You can also use conditional properties for finer control:

```
exe -a -: -a.cpp -b -;
obj -b -: -b.cpp -: -<variant>release:<optimization>off -;
```

Why are the `dll-path` and `hardcode-dll-paths` properties useful?

Note

This entry is specific to Unix systems.

Before answering the questions, let us recall a few points about shared libraries. Shared libraries can be used by several applications, or other libraries, without physically including the library in the application which can greatly decrease the total application size. It is also possible to upgrade a shared library when the application is already installed.

However, in order for application depending on shared libraries to be started the OS may need to find the shared library when the application is started. The dynamic linker will search in a system-defined list of paths, load the library and resolve the symbols. Which means that you should either change the system-defined list, given by the `LD_LIBRARY_PATH` environment variable, or install the libraries to a system location. This can be inconvenient when developing, since the libraries are not yet ready to be installed, and cluttering system paths may be undesirable. Luckily, on Unix there is another way.

An executable can include a list of additional library paths, which will be searched before system paths. This is excellent for development because the build system knows the paths to all libraries and can include them in the executables. That is done when the `hardcode-dll-paths` feature has the `true` value, which is the default. When the executables should be installed, the story is different.

Obviously, installed executable should not contain hardcoded paths to your development tree. (The `install` rule explicitly disables the `hardcode-dll-paths` feature for that reason.) However, you can use the `dll-path` feature to add explicit paths manually. For example:

```
install -installed -: -application -: -<dll-path>/usr/lib/snake
 - - -
```

paths while Solaris guidelines suggest that they should always be used.

Targets in site-config.jam

It is desirable to declare standard libraries available on a given system. Putting target declaration in a specific project's Jamfile is not really good, since locations of the libraries can vary between different development machines and then such declarations would need to be duplicated in different projects. The solution is to declare the targets in Boost.Build's `site-config.jam` configuration file:

```
project -site-config -;
lib -zlib -: -: -<name>z -;
```

Recall that both `site-config.jam` and `user-config.jam` are projects, and everything you can do in a Jamfile you can do in those files as well. So, you declare a project id and a target. Now, one can write:

```
exe -hello -: -hello.cpp -/site-config//zlib -;
```

in any Jamfile.

Header-only libraries

In modern C++, libraries often consist of just header files, without any source files to compile. To use such libraries, you need to add proper includes and possibly defines to your project. But with a large number of external libraries it becomes problematic to remember which libraries are header only, and which ones you have to link to. However, with Boost.Build a header-only library can be declared as Boost.Build target and all dependents can use such library without having to remeber whether it is a header-only library or not.

Header-only libraries may be declared using the `alias` rule, specifying their include path as a part of its usage requirements, for example:

```
alias -my-lib
 - - - -: -# -no -sources
 - - - -: -# -no -build -requirements
 - - - -: -# -no -default -build
 - - - -: -<include>whatever -;
```

The includes specified in usage requirements of `my-lib` are automatically added to all of its dependants' build properties. The dependants need not care if `my-lib` is a header-only or not, and it is possible to later make `my-lib` into a regular compiled library without having to that its dependants' declarations.

If you already have proper usage requirements declared for a project where a header-only library is defined, you do not need to duplicate them for the `alias` target:

```
project -my -: -usage-requirements -<include>whatever -;
alias -mylib -;
```

Appendix A. Boost.Jam Documentation

Introduction

Warning

Most probably, you are looking for Boost.Build manual. This document is not meant to be read standalone and will only confuse you. Boost.Build manual refers to specific sections when necessary.

Boost.Jam (BJam) is the low-level build engine tool for Boost.Build. Historically, Boost.Jam is based on on FTJam and on Perforce Jam but has grown a number of significant features and is now developed independently, with no merge back expected to happen, and little use outside Boost.Build.

This is version 3.1.18 of BJam and is based on version 2.4 of Jam/MR:

```
+\
+\ - -Copyright -1993-2002 -Christopher -Seiwald -and -Perforce -Software, -Inc.
\+
This -is -Release -2.4 -of -Jam/MR, -a -make-like -program.
License -is -hereby -granted -to -use -this -software -and -distribute -it
freely, -as -long -as -this -copyright -notice -is -retained -and -modifications
are -clearly -marked.
ALL -WARRANTIES -ARE -HEREBY -DISCLAIMED.
```

Building BJam

Installing `BJam` after building it is simply a matter of copying the generated executables someplace in your `PATH`. For building the executables there are a set of `build` bootstrap scripts to accomodate particular environments. The scripts take one optional argument, the name of the toolset to build with. When the toolset is not given an attempt is made to detect an available toolset and use that. The build scripts accept these arguments:

```
build -[toolset]
```

Running the scripts without arguments will give you the best chance of success. On Windows platforms from a command console do:

```
cd -jam -source -location
.\build.bat
```

On Unix type platforms do:

```
cd -jam -source -location
sh -./build.sh
```

For the Boost.Jam source included with the Boost distribution the *jam source location* is `BOOST_ROOT/tools/jam/src`.

If the scripts fail to detect an appropriate toolset to build with your particular toolset may not be auto-detectable. In that case, you can specify the toolset as the first argument, this assumes that the toolset is readily available in the `PATH`.

Note

The toolset used to build Boost.Jam is independent of the toolsets used for Boost.Build. Only one version of Boost.Jam is needed to use Boost.Build.

The supported toolsets, and whether they are auto-detected, are:

Table A.1. Supported Toolsets

Script	Platform	Toolset	Detection and Notes
`build.bat`	Windows NT, 2000, and XP	`borland` Borland C++Builder (BCC 5.5)	• Common install location: `"C:\Borland\BCC55"` • `BCC32.EXE` in `PATH`
		`como` Comeau Computing C/C++	
		`gcc` GNU GCC	
		`gcc-nocygwin` GNU GCC	
		`intel-win32` Intel C++ Compiler for Windows	• `ICL.EXE` in `PATH`
		`metrowerks` MetroWerks CodeWarrior C/C++ 7.x, 8.x, 9.x	• `CWFolder` variable configured • `MWCC.EXE` in `PATH`
		`mingw` GNU GCC as the MinGW configuration	• Common install location: `"C:\MinGW"`
		`msvc`	

Script	Platform	Toolset	Detection and Notes
		Microsoft Visual C++ 6.x	• VCVARS32.BAT already configured • %MSVCDir% is present in environment • Common install locations: %ProgramFiles%\Microsoft Visual Studio", %ProgramFiles%\Microsoft Visual C++" • CL.EXE in PATH
		vc7 Microsoft Visual C++ 7.x	• VCVARS32.BAT or VSVARS32.BAT already configured • %VS71COMNTOOLS% is present in environment • %VCINSTALLDIR% is present in environment • Common install locations: %ProgramFiles%\Microsoft Visual Studio .NET", %ProgramFiles%\Microsoft Visual Studio .NET 2003" • CL.EXE in PATH
		vc8 and vc9 Microsoft Visual C++ 8.x and 9.x	Detection: • VCVARSALL.BAT already configured • %VS90COMNTOOLS% is present in environment • Common install location: %ProgramFiles%\Microsoft Visual Studio 9" • %VS80COMNTOOLS% is present in environment • Common install location:

Script	Platform	Toolset	Detection and Notes
			%ProgramFiles%\Mic "rosoft Visual Studio 8" • CL.EXE in PATH Notes: • If VCVARSALL.BAT is called to set up the toolset, it is passed all the extra arguments, see below for what those arguments are. This can be used to build, for example, a Win64 specific version of bjam. Consult the VisualStudio documentation for what the possible argument values to the VCVARSALL.BAT are.
build.sh	Unix, Linux, Cygwin, etc.	acc HP-UX aCC	• aCC in PATH • uname is "HP-UX"
		como Comeau Computing C/C++	• como in PATH
		gcc GNU GCC	• gcc in PATH
		intel-linux Intel C++ for Linux	• icc in PATH • Common install locations: "/opt/intel/cc/9.0", "/opt/intel_cc_80", "/opt/intel/compiler70", "/opt/intel/compiler60", "/opt/intel/compiler50"

Script	Platform	Toolset	Detection and Notes
		`kcc` Intel KAI C++	• `KCC` in `PATH`
		`kylix` Borland C++Builder	• bc++ in PATH
		`mipspro` SGI MIPSpro C	• `uname` is `"IRIX"` or `"IRIX64"`
		`sunpro` Sun Workshop 6 C++	• Standard install location: `"/opt/SUNWspro"`
		`qcc` QNX Neutrino	• `uname` is `"QNX"` and `qcc` in `PATH`
		`true64cxx` Compaq C++ Compiler for True64 UNIX	• `uname` is `"OSF1"`
		`vacpp` IBM VisualAge C++	• `xlc` in `PATH`
	MacOS X	`darwin` Apple MacOS X GCC	• `uname` is `"Darwin"`
	Windows NT, 2000, and XP	`mingw` GNU GCC as the MinGW configuration with the MSYS shell	• Common install location: `"/mingw"`

The built executables are placed in a subdirectory specific to your platform. For example, in Linux running on an Intel x86 com-

patible chip, the executables are placed in: "`bin.linuxx86`". The `bjam[.exe]` executable can be used to invoke Boost.Build.

The build scripts support additional invocation arguments for use by developers of Boost.Jam and for additional setup of the toolset. The extra arguments come after the toolset:

- Arguments not in the form of an option, before option arguments, are used for extra setup to toolset configuration scripts.
- Arguments of the form "`--option`", which are passed to the `build.jam` build script.
- Arguments not in the form of an option, after the options, which are targets for the `build.jam` script.

```
build -[toolset] -[setup*] -[--option+ -target*]
```

The arguments immediately after the toolset are passed directly to the setup script of the toolset, if available and if it needs to be invoked. This allows one to configure the toolset ass needed to do non-default builds of `bjam`. For example to build a Win64 version with `vc8`. See the toolset descriptiona above for when particular toolsets support this.

The arguments starting with the "`--option`" forms are passed to the `build.jam` script and are used to further customize what gets built. Options and targets supported by the `build.jam` script:

`---`	Empty option when one wants to only specify a target.
`--release`	The default, builds the optimized executable.
`--debug`	Builds debugging versions of the executable. When built they are placed in their own directory "`bin./platform/.debug`".
`--grammar`	Normally the Jam language grammar parsing files are not regenerated. This forces building of the grammar, although it may not force the regeneration of the grammar parser. If the parser is out of date it will be regenerated and subsequently built.
`--with-python=`*`path`*	Enables Python integration, given a path to the Python libraries.
`--gc`	Enables use of the Boehm Garbage Collector. The build will look for the Boehm-GC source in a "boehm_gc" subdirectory from the `bjam` sources.
`--duma`	Enables use of the DUMA (Detect Uintended Memory Access) debugging memory allocator. The build expects to find the DUMA source files in a "duma" subdirectory from the `bjam` sources.
`--toolset-root=`*`path`*	Indicates where the toolset used to build is located. This option is passed in by the bootstrap (`build.bat` or `build.sh`) script.
`--show-locate-target`	For information, prints out where it will put the built executable.
`--noassert`	Disable debug assertions, even if building the debug version of the executable.
`dist`	Generate packages (compressed archives) as appropriate for distribution in the platform, if possible.
`clean`	Remove all the built executables and objects.

Using BJam

Warning

Most probably, you are looking for Boost.Build manual or Boost.Build command-line syntax. This section documents only low-level options used by the Boost.Jam build engine, and does not mention any high-level syntax of Boost.Build

If *target* is provided on the command line, `bjam` builds *target*; otherwise `bjam` builds the target `all`.

```
bjam -( --option -[value] -| -target -) -*
```

Options

Options are either singular or have an accompanying value. When a value is allowed, or required, it can be either given as an argument following the option argument, or it can be given immediately after the option as part of the option argument. The allowed options are:

`-a` Build all targets anyway, even if they are up-to-date.

`-d n` Enable cummulative debugging levels from 1 to n. Values are:

1. Show the actions taken for building targets, as they are executed (the default).
2. Show "quiet" actions and display all action text, as they are executed.
3. Show dependency analysis, and target/source timestamps/paths.
4. Show arguments and timming of shell invocations.
5. Show rule invocations and variable expansions.
6. Show directory/header file/archive scans, and attempts at binding to targets.
7. Show variable settings.
8. Show variable fetches, variable expansions, and evaluation of "'if"' expressions.
9. Show variable manipulation, scanner tokens, and memory usage.
10. Show profile information for rules, both timing and memory.
11. Show parsing progress of Jamfiles.
12. Show graph of target dependencies.
13. Show change target status (fate).

`-d +n` Enable debugging level *n*.

`-d 0` Turn off all debugging levels. Only errors are reported.

`-f Jambase` Read *Jambase* instead of using the built-in Jambase. Only one -f flag is permitted, but the *Jambase* may explicitly include other files. A *Jambase* name of "-" is allowed, in which case console input is read until it is closed, at which point the input is treated as the Jambase.

`-j n` Run up to *n* shell commands concurrently (UNIX and NT only). The default is 1.

`-l` *n*	Limit actions to running for *n* number of seconds, after which they are stopped. Note: Windows only.
`-n`	Don't actually execute the updating actions, but do everything else. This changes the debug level default to `-d 2`.
`-o` *file*	Write the updating actions to the specified file instead of running them.
`-q`	Quit quickly (as if an interrupt was received) as soon as **any** target fails.
`-s` *var=value*	Set the variable *var* to *value*, overriding both internal variables and variables imported from the environment.
`-t` *target*	Rebuild *target* and everything that depends on it, even if it is up-to-date.
`--` *value*	The option and *value* is ignored, but is available from the `$(ARGV)` variable.
`-v`	Print the version of `bjam` and exit.

Command-line and Environment Variable Quoting

Classic Jam had an odd behavior with respect to command-line variable (`-s...`) and environment variable settings which made it impossible to define an arbitrary variable with spaces in the value. Boost Jam remedies that by treating all such settings as a single string if they are surrounded by double-quotes. Uses of this feature can look interesting, since shells require quotes to keep characters separated by whitespace from being treated as separate arguments:

```
jam --sMSVCNT="\"\"C:\Program -Files\Microsoft -Visual -C++\VC98\"\"" -...
```

The outer quote is for the shell. The middle quote is for Jam, to tell it to take everything within those quotes literally, and the inner quotes are for the shell again when paths are passed as arguments to build actions. Under NT, it looks a lot more sane to use environment variables before invoking jam when you have to do this sort of quoting:

```
set -MSVCNT=""C:\Program -Files\Microsoft -Visual -C++\VC98\""
```

Operation

BJam has four phases of operation: start-up, parsing, binding, and updating.

Start-up

Upon start-up, `bjam` imports environment variable settings into `bjam` variables. Environment variables are split at blanks with each word becoming an element in the variable's list of values. Environment variables whose names end in `PATH` are split at `$(SPLITPATH)` characters (e.g., `":"` for Unix).

To set a variable's value on the command line, overriding the variable's environment value, use the `-s` option. To see variable assignments made during bjam's execution, use the `-d+7` option.

The Boost.Build v2 initialization behavior has been implemented. This behavior only applies when the executable being invoked is called "`bjam`" or, for backward-compatibility, when the `BOOST_ROOT` variable is set.

1. We attempt to load "`boost-build.jam`" by searching from the current invocation directory up to the root of the file system. This file is expected to invoke the `boost-build` rule to indicate where the Boost.Build system files are, and to load them.

2. If `boost-build.jam` is not found we error and exit, giving brief instructions on possible errors. As a backward-compatibility measure for older versions of Boost.Build, when the `BOOST_ROOT` variable is set, we first search for `boost-build.jam` in `$(BOOST_ROOT)/tools/build` and `$(BOOST_BUILD_PATH)`. If found, it is loaded and initialization is complete.

3. The `boost-build` rule adds its (optional) argument to the front of `BOOST_BUILD_PATH`, and attempts to load `boot-`

`strap.jam` from those directories. If a relative path is specified as an argument, it is treated as though it was relative to the `boost-build.jam` file.

4. If the `bootstrap.jam` file was not found, we print a likely error message and exit.

Parsing

In the parsing phase, `bjam` reads and parses the `Jambase` file, by default the built-in one. It is written in the jam language. The last action of the `Jambase` is to read (via the "include" rule) a user-provided file called "`Jamfile`".

Collectively, the purpose of the `Jambase` and the `Jamfile` is to name build targets and source files, construct the dependency graph among them, and associate build actions with targets. The `Jambase` defines boilerplate rules and variable assignments, and the `Jamfile` uses these to specify the actual relationship among the target and source files.

Binding

After parsing, `bjam` recursively descends the dependency graph and binds every file target with a location in the filesystem. If `bjam` detects a circular dependency in the graph, it issues a warning.

File target names are given as absolute or relative path names in the filesystem. If the path name is absolute, it is bound as is. If the path name is relative, it is normally bound as is, and thus relative to the current directory. This can be modified by the settings of the `$(SEARCH)` and `$(LOCATE)` variables, which enable jam to find and build targets spread across a directory tree. See SEARCH and LOCATE Variables below.

Update Determination

After binding each target, `bjam` determines whether the target needs updating, and if so marks the target for the updating phase. A target is normally so marked if it is missing, it is older than any of its sources, or any of its sources are marked for updating. This behavior can be modified by the application of special built-in rules, `ALWAYS`, `LEAVES`, `NOCARE`, `NOTFILE`, `NOUPDATE`, and `TEMPORARY`. See Modifying Binding below.

Header File Scanning

During the binding phase, `bjam` also performs header file scanning, where it looks inside source files for the implicit dependencies on other files caused by C's #include syntax. This is controlled by the special variables $(HDRSCAN) and $(HDRRULE). The result of the scan is formed into a rule invocation, with the scanned file as the target and the found included file names as the sources. Note that this is the only case where rules are invoked outside the parsing phase. See HDRSCAN and HDRRULE Variables below.

Updating

After binding, `bjam` again recursively descends the dependency graph, this time executing the update actions for each target marked for update during the binding phase. If a target's updating actions fail, then all other targets which depend on that target are skipped.

The `-j` flag instructs `bjam` to build more than one target at a time. If there are multiple actions on a single target, they are run sequentially.

Language

`BJam` has an interpreted, procedural language. Statements in `bjam` are rule (procedure) definitions, rule invocations, flow-of-control structures, variable assignments, and sundry language support.

Lexical Features

`BJam` treats its input files as whitespace-separated tokens, with two exceptions: double quotes (") can enclose whitespace to embed it into a token, and everything between the matching curly braces ({}) in the definition of a rule action is treated as a single string. A backslash (\) can escape a double quote, or any single whitespace character.

BJam requires whitespace (blanks, tabs, or newlines) to surround all tokens, including the colon (:) and semicolon (;) tokens.

BJam keywords (an mentioned in this document) are reserved and generally must be quoted with double quotes (") to be used as arbitrary tokens, such as variable or target names.

Comments start with the # character and extend until the end of line.

Targets

The essential bjam data entity is a target. Build targets are files to be updated. Source targets are the files used in updating built targets. Built targets and source targets are collectively referred to as file targets, and frequently built targets are source targets for other built targets. Pseudotargets are symbols which represent dependencies on other targets, but which are not themselves associated with any real file.

A file target's identifier is generally the file's name, which can be absolutely rooted, relative to the directory of bjam's invocation, or simply local (no directory). Most often it is the last case, and the actual file path is bound using the $(SEARCH) and $(LOCATE) special variables. See SEARCH and LOCATE Variables below. A local filename is optionally qualified with grist, a string value used to assure uniqueness. A file target with an identifier of the form *file(member)* is a library member (usually an ar(1) archive on Unix).

Binding Detection

Whenever a target is bound to a location in the filesystem, Boost Jam will look for a variable called BINDRULE (first "on" the target being bound, then in the global module). If non-empty, =$(BINDRULE[1])= names a rule which is called with the name of the target and the path it is being bound to. The signature of the rule named by =$(BINDRULE[1])= should match the following:

```
rule -bind-rule -( -target -: -path -)
```

This facility is useful for correct header file scanning, since many compilers will search for #include files first in the directory containing the file doing the #include directive. $(BINDRULE) can be used to make a record of that directory.

Rules

The basic bjam language entity is called a rule. A rule is defined in two parts: the procedure and the actions. The procedure is a body of jam statements to be run when the rule is invoked; the actions are the OS shell commands to execute when updating the built targets of the rule.

Rules can return values, which can be expanded into a list with "[*rule args ...*]". A rule's value is the value of its last statement, though only the following statements have values: 'if' (value of the leg chosen), 'switch' (value of the case chosen), set (value of the resulting variable), and 'return' (value of its arguments). Note that 'return' doesn't actually cause a return, i.e., is a no-op unless it is the last statement of the last block executed within rule body.

The bjam statements for defining and invoking rules are as follows:

Define a rule's procedure, replacing any previous definition.

```
rule -rulename -{ -statements -}
```

Define a rule's updating actions, replacing any previous definition.

```
actions -[ -modifiers -] -rulename -{ -commands -}
```

Invoke a rule.

```
rulename -field1 -: -field2 -: -... -: -fieldN -;
```

Invoke a rule under the influence of target's specific variables..

```
on -target -rulename -field1 -: -field2 -: -... -: -fieldN -;
```

Used as an argument, expands to the return value of the rule invoked.

```
[ -rulename -field1 -: -field2 -: -... -: -fieldN -]
[ -on -target -rulename -field1 -: -field2 -: -... -: -fieldN -]
```

A rule is invoked with values in *field1* through *fieldN*. They may be referenced in the procedure's statements as `$(1)` through `$(N)` (9 max), and the first two only may be referenced in the action's *commands* as `$(1)` and `$(2)`. `$(<)` and `$(>)` are synonymous with `$(1)` and `$(2)`.

Rules fall into two categories: updating rules (with actions), and pure procedure rules (without actions). Updating rules treat arguments `$(1)` and `$(2)` as built targets and sources, respectively, while pure procedure rules can take arbitrary arguments.

When an updating rule is invoked, its updating actions are added to those associated with its built targets (`$(1)`) before the rule's procedure is run. Later, to build the targets in the updating phase, *commands* are passed to the OS command shell, with `$(1)` and `$(2)` replaced by bound versions of the target names. See Binding above.

Rule invocation may be indirected through a variable:

```
$(var) -field1 -: -field2 -: -... -: -fieldN -;

on -target -$(var) -field1 -: -field2 -: -... -: -fieldN -;

[ -$(var) -field1 -: -field2 -: -... -: -fieldN -]
[ -on -target -$(var) -field1 -: -field2 -: -... -: -fieldN -]
```

The variable's value names the rule (or rules) to be invoked. A rule is invoked for each element in the list of `$(var)`'s values. The fields *field1* : *field2* : ... are passed as arguments for each invokation. For the [...] forms, the return value is the concatenation of the return values for all of the invocations.

Action Modifiers

The following action modifiers are understood:

`actions bind vars`	`$(vars)` will be replaced with bound values.
`actions existing`	`$(>)` includes only source targets currently existing.
`actions ignore`	The return status of the commands is ignored.
`actions piecemeal`	commands are repeatedly invoked with a subset of `$(>)` small enough to fit in the command buffer on this OS.
`actions quietly`	The action is not echoed to the standard output.
`actions together`	The `$(>)` from multiple invocations of the same action on the same built target are glommed together.
`actions updated`	`$(>)` includes only source targets themselves marked for updating.

Argument lists

You can describe the arguments accepted by a rule, and refer to them by name within the rule. For example, the following prints

"I'm sorry, Dave" to the console:

```
rule -report -( -pronoun -index -? -: -state -: -names -+ -)
{
 - - - -local -he.suffix -she.suffix -it.suffix -= -s -;
 - - - -local -I.suffix -= -m -;
 - - - -local -they.suffix -you.suffix -= -re -;
 - - - -ECHO -$(pronoun)'$($(pronoun).suffix) -$(state), -$(names[$(index)]) -;
}
report -I -2 -: -sorry -: -Joe -Dave -Pete -;
```

Each name in a list of formal arguments (separated by ":" in the rule declaration) is bound to a single element of the corresponding actual argument unless followed by one of these modifiers:

Symbol	**Semantics of preceding symbol**
?	optional
*	Bind to zero or more unbound elements of the actual argument. When * appears where an argument name is expected, any number of additional arguments are accepted. This feature can be used to implement "varargs" rules.
+	Bind to one or more unbound elements of the actual argument.

The actual and formal arguments are checked for inconsistencies, which cause Jam to exit with an error code:

```
### -argument -error
# -rule -report -( -pronoun -index -? - -: -state - -: -names -+ -)
# -called -with: -( -I -2 -foo - -: -sorry - -: -Joe -Dave -Pete -)
# -extra -argument -foo
### -argument -error
# -rule -report -( -pronoun -index -? - -: -state - -: -names -+ -)
# -called -with: -( -I -2 - -: -sorry -)
# -missing -argument -names
```

If you omit the list of formal arguments, all checking is bypassed as in "classic" Jam. Argument lists drastically improve the reliability and readability of your rules, however, and are **strongly recommended** for any new Jam code you write.

Built-in Rules

BJam has a growing set of built-in rules, all of which are pure procedure rules without updating actions. They are in three groups: the first builds the dependency graph; the second modifies it; and the third are just utility rules.

Dependency Building

DEPENDS

```
rule -DEPENDS -( -targets1 -* -: -targets2 -* -)
```

Builds a direct dependency: makes each of *targets1* depend on each of *targets2*. Generally, *targets1* will be rebuilt if *targets2* are themselves rebuilt or are newer than *targets1*.

INCLUDES

```
rule -INCLUDES -( -targets1 -* -: -targets2 -* -)
```

Builds a sibling dependency: makes any target that depends on any of *targets1* also depend on each of *targets2*. This reflects the dependencies that arise when one source file includes another: the object built from the source file depends both on the original and included source file, but the two sources files don't depend on each other. For example:

```
DEPENDS -foo.o -: -foo.c -;
INCLUDES -foo.c -: -foo.h -;
```

"`foo.o`" depends on "`foo.c`" and "`foo.h`" in this example.

Modifying Binding

The six rules `ALWAYS`, `LEAVES`, `NOCARE`, `NOTFILE`, `NOUPDATE`, and `TEMPORARY` modify the dependency graph so that `bjam` treats the targets differently during its target binding phase. See Binding above. Normally, `bjam` updates a target if it is missing, if its filesystem modification time is older than any of its dependencies (recursively), or if any of its dependencies are being updated. This basic behavior can be changed by invoking the following rules:

ALWAYS

```
rule -ALWAYS -( -targets -* -)
```

Causes *targets* to be rebuilt regardless of whether they are up-to-date (they must still be in the dependency graph). This is used for the clean and uninstall targets, as they have no dependencies and would otherwise appear never to need building. It is best applied to targets that are also `NOTFILE` targets, but it can also be used to force a real file to be updated as well.

LEAVES

```
rule -LEAVES -( -targets -* -)
```

Makes each of *targets* depend only on its leaf sources, and not on any intermediate targets. This makes it immune to its dependencies being updated, as the "leaf" dependencies are those without their own dependencies and without updating actions. This allows a target to be updated only if original source files change.

NOCARE

```
rule -NOCARE -( -targets -* -)
```

Causes `bjam` to ignore *targets* that neither can be found nor have updating actions to build them. Normally for such targets `bjam` issues a warning and then skips other targets that depend on these missing targets. The `HdrRule` in `Jambase` uses `NOCARE` on the header file names found during header file scanning, to let `bjam` know that the included files may not exist. For example, if an `#include` is within an `#ifdef`, the included file may not actually be around.

Warning

For targets with build actions: if their build actions exit with a nonzero return code, dependent targets will still be built.

NOTFILE

```
rule -NOTFILE -( -targets -* -)
```

Marks *targets* as pseudotargets and not real files. No timestamp is checked, and so the actions on such a target are only executed if the target's dependencies are updated, or if the target is also marked with `ALWAYS`. The default `bjam` target "`all`" is a pseudotarget. In `Jambase`, `NOTFILE` is used to define several addition convenient pseudotargets.

NOUPDATE

```
rule -NOUPDATE -( -targets -* -)
```

Causes the timestamps on *targets* to be ignored. This has two effects: first, once the target has been created it will never be updated; second, manually updating target will not cause other targets to be updated. In `Jambase`, for example, this rule is applied to directories by the `MkDir` rule, because `MkDir` only cares that the target directory exists, not when it has last been updated.

TEMPORARY

```
rule -TEMPORARY -( -targets -* -)
```

Marks *targets* as temporary, allowing them to be removed after other targets that depend upon them have been updated. If a `TEMPORARY` target is missing, `bjam` uses the timestamp of the target's parent. `Jambase` uses `TEMPORARY` to mark object files that are archived in a library after they are built, so that they can be deleted after they are archived.

FAIL_EXPECTED

```
rule -FAIL_EXPECTED -( -targets -* -)
```

For handling targets whose build actions are expected to fail (e.g. when testing that assertions or compile-time type checking work properly), Boost Jam supplies the `FAIL_EXPECTED` rule in the same style as `NOCARE`, et. al. During target updating, the return code of the build actions for arguments to `FAIL_EXPECTED` is inverted: if it fails, building of dependent targets continues as though it succeeded. If it succeeds, dependent targets are skipped.

RMOLD

```
rule -RMOLD -( -targets -* -)
```

`BJam` removes any target files that may exist on disk when the rule used to build those targets fails. However, targets whose dependencies fail to build are not removed by default. The `RMOLD` rule causes its arguments to be removed if any of their dependencies fail to build.

ISFILE

```
rule -ISFILE -( -targets -* -)
```

`ISFILE` marks targets as required to be files. This changes the way `bjam` searches for the target such that it ignores mathes for file system items that are not file, like directories. This makes it possible to avoid `#include "exception"` matching if one happens to have a directory named exception in the header search path.

Warning

This is currently not fully implemented.

Utility

The two rules ECHO and EXIT are utility rules, used only in bjam's parsing phase.

ECHO

```
rule -ECHO -( -args -* -)
```

Blurts out the message *args* to stdout.

EXIT

```
rule -EXIT -( -message -* -: -result-value -? -)
```

Blurts out the *message* to stdout and then exits with a failure status if no *result-value* is given, otherwise it exits with the given *result-value*.

"Echo", "echo", "Exit", and "exit" are accepted as aliases for ECHO and EXIT, since it is hard to tell that these are built-in rules and not part of the language, like "include".

GLOB

The GLOB rule does filename globbing.

```
rule -GLOB -( -directories -* -: -patterns -* -: -downcase-opt -? -)
```

Using the same wildcards as for the patterns in the switch statement. It is invoked by being used as an argument to a rule invocation inside of "[ ]". For example: "FILES = [GLOB dir1 dir2 : *.c *.h]" sets FILES to the list of C source and header files in dir1 and dir2. The resulting filenames are the full pathnames, including the directory, but the pattern is applied only to the file name without the directory.

If *downcase-opt* is supplied, filenames are converted to all-lowercase before matching against the pattern; you can use this to do case-insensitive matching using lowercase patterns. The paths returned will still have mixed case if the OS supplies them. On Windows NT and Cygwin, filenames are always downcased before matching.

MATCH

The MATCH rule does pattern matching.

```
rule -MATCH -( -regexps -+ -: -list -* -)
```

Matches the egrep(1) style regular expressions *regexps* against the strings in *list*. The result is the concatenation of matching () subexpressions for each string in *list*, and for each regular expression in *regexps*. Only useful within the "[ ]" construct, to change the result into a list.

BACKTRACE

```
rule -BACKTRACE -( -)
```

Returns a list of quadruples: *filename line module rulename...*, describing each shallower level of the call stack. This rule can be used to generate useful diagnostic messages from Jam rules.

UPDATE

```
rule -UPDATE -( -targets -* -)
```

Classic jam treats any non-option element of command line as a name of target to be updated. This prevented more sophisticated handling of command line. This is now enabled again but with additional changes to the `UPDATE` rule to allow for the flexibility of changing the list of targets to update. The UPDATE rule has two effects:

1. It clears the list of targets to update, and
2. Causes the specified targets to be updated.

If no target was specified with the `UPDATE` rule, no targets will be updated. To support changing of the update list in more useful ways, the rule also returns the targets previously in the update list. This makes it possible to add targets as such:

```
local -previous-updates -= -[ -UPDATE -] -;
UPDATE -$(previous-updates) -a-new-target -;
```

W32_GETREG

```
rule -W32_GETREG -( -path -: -data -? -)
```

Defined only for win32 platform. It reads the registry of Windows. '*path*' is the location of the information, and '*data*' is the name of the value which we want to get. If '*data*' is omitted, the default value of '*path*' will be returned. The '*path*' value must conform to MS key path format and must be prefixed with one of the predefined root keys. As usual,

- '`HKLM`' is equivalent to '`HKEY_LOCAL_MACHINE`'.
- '`HKCU`' is equivalent to '`HKEY_CURRENT_USER`'.
- '`HKCR`' is equivalent to '`HKEY_CLASSES_ROOT`'.

Other predefined root keys are not supported.

Currently supported data types : '`REG_DWORD`', '`REG_SZ`', '`REG_EXPAND_SZ`', '`REG_MULTI_SZ`'. The data with '`REG_DWORD`' type will be turned into a string, '`REG_MULTI_SZ`' into a list of strings, and for those with '`REG_EXPAND_SZ`' type environment variables in it will be replaced with their defined values. The data with '`REG_SZ`' type and other unsupported types will be put into a string without modification. If it can't receive the value of the data, it just return an empty list. For example,

```
local -PSDK-location -=
 - -[ -W32_GETREG -HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\MicrosoftSDK\\Directories -: -"Ins
tall -Dir" -] -;
```

W32_GETREGNAMES

```
rule -W32_GETREGNAMES -( -path -: -result-type -)
```

Defined only for win32 platform. It reads the registry of Windows. '*path*' is the location of the information, and '*result-type*' is either '`subkeys`' or '`values`'. For more information on '*path*' format and constraints, please see `W32_GETREG`.

Depending on '*result-type*', the rule returns one of the following:

`subkeys` Names of all direct subkeys of '*path*'.

`values` Names of values contained in registry key given by '*path*'. The "default" value of the key appears in the returned list only if its value has been set in the registry.

If '*result-type*' is not recognized, or requested data cannot be retrieved, the rule returns an empty list. Example:

```
loc#
al -key -= -"HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\App -Paths" -;
local -subkeys -= -[ -W32_GETREGNAMES -"$(key)" -: -subkeys -] -;
for -local -subkey -in -$(subkeys)
{
 - - - -local -values -= -[ -W32_GETREGNAMES -"$(key)\\$(subkey)" -: -values -] -;
 - - - -for -local -value -in -$(values)
 - - - -{
 - - - - - - - - -local -data -= -[ -W32_GETREG -"$(key)\\$(subkey)" -: -"$(value)" -] -;
 - - - - - - - - -ECHO -"Registry -path: -" -$(key)\\$(subkey) -":" -$(value) -"=" -$(data) -;
 - - - -}
}
```

SHELL

```
rule -SHELL -( -command -: -* -)
```

`SHELL` executes *command*, and then returns the standard output of *command*. `SHELL` only works on platforms with a `popen()` function in the C library. On platforms without a working `popen()` function, `SHELL` is implemented as a no-op. `SHELL` works on Unix, MacOS X, and most Windows compilers. `SHELL` is a no-op on Metrowerks compilers under Windows. There is a variable set of allowed options as additional arguments:

`exit-status` In addition to the output the result status of the executed command is returned as a second element of the result.

`no-output` Don't capture the output of the command. Instead an empty ("") string value is returned in place of the output.

Because the Perforce/Jambase defines a `SHELL` rule which hides the builtin rule, `COMMAND` can be used as an alias for `SHELL` in such a case.

Flow-of-Control

`BJam` has several simple flow-of-control statements:

```
for -var -in -list -{ -statements -}
```

Executes *statements* for each element in *list*, setting the variable *var* to the element value.

```
if -cond -{ -statements -}
[ -else -{ -statements -} -]
```

Does the obvious; the `else` clause is optional. *cond* is built of:

a true if any *a* element is a non-zero-length string

a = b list *a* matches list *b* string-for-string

`a != b`	list *a* does not match list *b*
`a < b`	*a[i]* string is less than *b[i]* string, where *i* is first mismatched element in lists *a* and *b*
`a <= b`	every *a* string is less than or equal to its *b* counterpart
`a > b`	*a[i]* string is greater than *b[i]* string, where *i* is first mismatched element
`a >= b`	every *a* string is greater than or equal to its *b* counterpart
`a in b`	true if all elements of *a* can be found in *b*, or if *a* has no elements
`! cond`	condition not true
`cond && cond`	conjunction
`cond || cond`	disjunction
`( cond )`	precedence grouping

```
include -file -;
```

Causes `bjam` to read the named *file*. The *file* is bound like a regular target (see Binding above) but unlike a regular target the include *file* cannot be built.

The include *file* is inserted into the input stream during the parsing phase. The primary input file and all the included file(s) are treated as a single file; that is, jam infers no scope boundaries from included files.

```
local -vars -[ -= -values -] -;
```

Creates new *vars* inside to the enclosing `{}` block, obscuring any previous values they might have. The previous values for vars are restored when the current block ends. Any rule called or file included will see the local and not the previous value (this is sometimes called Dynamic Scoping). The local statement may appear anywhere, even outside of a block (in which case the previous value is restored when the input ends). The *vars* are initialized to *values* if present, or left uninitialized otherwise.

```
return -values -;
```

Within a rule body, the return statement sets the return value for an invocation of the rule. It does **not** cause the rule to return; a rule's value is actually the value of the last statement executed, so a return should be the last statement executed before the rule "naturally" returns.

```
switch -value
{
 - - - -case -pattern1 -: -statements -;
 - - - -case -pattern2 -: -statements -;
 - - - -...
}
```

The switch statement executes zero or one of the enclosed *statements*, depending on which, if any, is the first case whose *pattern* matches *value*. The *pattern* values are not variable-expanded. The pattern values may include the following wildcards:

`?`	match any single character
`*`	match zero or more characters

`[chars]` match any single character in *chars*

`[^chars]` match any single character not in *chars*

`\x` match *x* (escapes the other wildcards)

```
while -cond -{ -statements -}
```

Repeatedly execute *statements* while *cond* remains true upon entry. (See the description of *cond* expression syntax under if, above).

Variables

`BJam` variables are lists of zero or more elements, with each element being a string value. An undefined variable is indistinguishable from a variable with an empty list, however, a defined variable may have one more elements which are null strings. All variables are referenced as `$(variable)`.

Variables are either global or target-specific. In the latter case, the variable takes on the given value only during the updating of the specific target.

A variable is defined with:

```
variable -= -elements -;
variable -+= -elements -;
variable -on -targets -= -elements -;
variable -on -targets -+= -elements -;
variable -default -= -elements -;
variable -?= -elements -;
```

The first two forms set *variable* globally. The third and forth forms set a target-specific variable. The `=` operator replaces any previous elements of *variable* with *elements*; the `+=` operation adds *elements* to *variable*'s list of elements. The final two forms are synonymous: they set *variable* globally, but only if it was previously unset.

Variables referenced in updating commands will be replaced with their values; target-specific values take precedence over global values. Variables passed as arguments (`$(1)` and `$(2)`) to actions are replaced with their bound values; the "`bind`" modifier can be used on actions to cause other variables to be replaced with bound values. See Action Modifiers above.

`BJam` variables are not re-exported to the environment of the shell that executes the updating actions, but the updating actions can reference `bjam` variables with `$(variable)`.

Variable Expansion

During parsing, `bjam` performs variable expansion on each token that is not a keyword or rule name. Such tokens with embedded variable references are replaced with zero or more tokens. Variable references are of the form `$(v)` or `$(vm)`, where *v* is the variable name, and *m* are optional modifiers.

Variable expansion in a rule's actions is similar to variable expansion in statements, except that the action string is tokenized at whitespace regardless of quoting.

The result of a token after variable expansion is the *product* of the components of the token, where each component is a literal substring or a list substituting a variable reference. For example:

```
$(X) --> -a -b -c
t$(X) --> -ta -tb -tc
$(X)z --> -az -bz -cz
$(X)-$(X) --> -a-a -a-b -a-c -b-a -b-b -b-c -c-a -c-b -c-c
```

The variable name and modifiers can themselves contain a variable reference, and this partakes of the product as well:

```
$(X) --> -a -b -c
$(Y) --> -1 -2
$(Z) --> -X -Y
$($(Z)) --> -a -b -c -1 -2
```

Because of this product expansion, if any variable reference in a token is undefined, the result of the expansion is an empty list. If any variable element is a null string, the result propagates the non-null elements:

```
$(X) --> -a -""
$(Y) --> -"" -1
$(Z) -->
-$(X)$(Y)- --> --a- --a1- --- --1-
-$(X)$(Z)- -->
```

A variable element's string value can be parsed into grist and filename-related components. Modifiers to a variable are used to select elements, select components, and replace components. The modifiers are:

`[n]` Select element number *n* (starting at 1). If the variable contains fewer than *n* elements, the result is a zero-element list. *n* can be negative in which case the element number *n* from the last leftward is returned.

`[n-m]` Select elements number *n* through *m*. *n* and *m* can be negative in which case they refer to elements counting from the last leftward.

`[n-]` Select elements number *n* through the last. *n* can be negative in which case it refers to the element counting from the last leftward.

`:B` Select filename base.

`:S` Select (last) filename suffix.

`:M` Select archive member name.

`:D` Select directory path.

`:P` Select parent directory.

`:G` Select grist.

`:U` Replace lowercase characters with uppercase.

`:L` Replace uppercase characters with lowercase.

`:T` Converts all back-slashes ("\") to forward slashes ("/"). For example

```
x -= -"C:\\Program -Files\\Borland" -; -ECHO -$(x:T) -;
```

prints `"C:/Program Files/Borland"`

`:W` When invoking Windows-based tools from Cygwin it can be important to pass them true windows-style paths. The `:W` modifier, **under Cygwin only**, turns a cygwin path into a Win32 path using the `cygwin_conv_to_win32_path` function. On other platforms, the string is unchanged. For example

```
x -= -"/cygdrive/c/Program -Files/Borland" -; -ECHO -$(x:W) -;
```

prints `"C:\Program Files\Borland"` on Cygwin

`:chars` Select the components listed in *chars*.

`:G=grist`	Replace grist with *grist*.
`:D=path`	Replace directory with *path*.
`:B=base`	Replace the base part of file name with *base*.
`:S=suf`	Replace the suffix of file name with *suf*.
`:M=mem`	Replace the archive member name with *mem*.
`:R=root`	Prepend *root* to the whole file name, if not already rooted.
`:E=value`	Assign *value* to the variable if it is unset.
`:J=joinval`	Concatentate list elements into single element, separated by *joinval*'.

On VMS, `$(var:P)` is the parent directory of `$(var:D)`.

Local For Loop Variables

Boost Jam allows you to declare a local for loop control variable right in the loop:

```
x -= -1 -2 -3 -;
y -= -4 -5 -6 -;
for -local -y -in -$(x)
{
 - - - -ECHO -$(y) -; -# -prints -"1", -"2", -or -"3"
}
ECHO -$(y) -; - - - - -# -prints -"4 -5 -6"
```

Generated File Expansion

During expansion of expressions `bjam` also looks for subexpressions of the form =@(filename:E`filecontents`) and replaces the expression with `filename` after creating the given file with the contents set to `filecontents`. This is useful for creating compiler response files, and other "internal" files. The expansion works both during parsing and action execution. Hence it is possible to create files during any of the three build phases.

Built-in Variables

This section discusses variables that have special meaning to `bjam`. All of these must be defined or used in the global module -- using those variables inside a named module will not have the desired effect. See Modules.

SEARCH and LOCATE

These two variables control the binding of file target names to locations in the file system. Generally, `$(SEARCH)` is used to find existing sources while `$(LOCATE)` is used to fix the location for built targets.

Rooted (absolute path) file targets are bound as is. Unrooted file target names are also normally bound as is, and thus relative to the current directory, but the settings of `$(LOCATE)` and `$(SEARCH)` alter this:

- If `$(LOCATE)` is set then the target is bound relative to the first directory in `$(LOCATE)`. Only the first element is used for binding.
- If `$(SEARCH)` is set then the target is bound to the first directory in `$(SEARCH)` where the target file already exists.
- If the `$(SEARCH)` search fails, the target is bound relative to the current directory anyhow.

Both `$(SEARCH)` and `$(LOCATE)` should be set target-specific and not globally. If they were set globally, `bjam` would use the same paths for all file binding, which is not likely to produce sane results. When writing your own rules, especially ones not built

upon those in Jambase, you may need to set `$(SEARCH)` or `$(LOCATE)` directly. Almost all of the rules defined in Jambase set `$(SEARCH)` and `$(LOCATE)` to sensible values for sources they are looking for and targets they create, respectively.

HDRSCAN and HDRRULE

These two variables control header file scanning. `$(HDRSCAN)` is an `egrep(1)` pattern, with ()'s surrounding the file name, used to find file inclusion statements in source files. `Jambase` uses `$(HDRPATTERN)` as the pattern for `$(HDRSCAN)`. `$(HDRRULE)` is the name of a rule to invoke with the results of the scan: the scanned file is the target, the found files are the sources. This is the only place where `bjam` invokes a rule through a variable setting.

Both `$(HDRSCAN)` and `$(HDRRULE)` must be set for header file scanning to take place, and they should be set target-specific and not globally. If they were set globally, all files, including executables and libraries, would be scanned for header file include statements.

The scanning for header file inclusions is not exact, but it is at least dynamic, so there is no need to run something like `makedepend(GNU)` to create a static dependency file. The scanning mechanism errs on the side of inclusion (i.e., it is more likely to return filenames that are not actually used by the compiler than to miss include files) because it can't tell if `#include` lines are inside `#ifdefs` or other conditional logic. In `Jambase`, `HdrRule` applies the `NOCARE` rule to each header file found during scanning so that if the file isn't present yet doesn't cause the compilation to fail, `bjam` won't care.

Also, scanning for regular expressions only works where the included file name is literally in the source file. It can't handle languages that allow including files using variable names (as the `Jam` language itself does).

Semaphores

It is sometimes desirable to disallow parallel execution of some actions. For example:

- Old versions of yacc use files with fixed names. So, running two yacc actions is dangerous.
- One might want to perform parallel compiling, but not do parallel linking, because linking is i/o bound and only gets slower.

Craig McPeeters has extended Perforce Jam to solve such problems, and that extension was integrated in Boost.Jam.

Any target can be assigned a *semaphore*, by setting a variable called `SEMAPHORE` on that target. The value of the variable is the semaphore name. It must be different from names of any declared target, but is arbitrary otherwise.

The semantic of semaphores is that in a group of targets which have the same semaphore, only one can be updated at the moment, regardless of "`-j`" option.

Platform Identifier

A number of Jam built-in variables can be used to identify runtime platform:

`OS`	OS identifier string
`OSPLAT`	Underlying architecture, when applicable
`MAC`	true on MAC platform
`NT`	true on NT platform
`OS2`	true on OS2 platform
`UNIX`	true on Unix platforms
`VMS`	true on VMS platform

Jam Version

JAMDATE	Time and date at bjam start-up as an ISO-8601 UTC value.
JAMUNAME	Ouput of uname(1) command (Unix only)
JAMVERSION	bjam version, currently "3.1.18"
JAM_VERSION	A predefined global variable with two elements indicates the version number of Boost Jam. Boost Jam versions start at "03" "00". Earlier versions of Jam do not automatically define JAM_VERSION.

JAMSHELL

When bjam executes a rule's action block, it forks and execs a shell, passing the action block as an argument to the shell. The invocation of the shell can be controlled by $(JAMSHELL). The default on Unix is, for example:

```
JAMSHELL -= -/bin/sh --c -% -;
```

The % is replaced with the text of the action block.

BJam does not directly support building in parallel across multiple hosts, since that is heavily dependent on the local environment. To build in parallel across multiple hosts, you need to write your own shell that provides access to the multiple hosts. You then reset $(JAMSHELL) to reference it.

Just as bjam expands a % to be the text of the rule's action block, it expands a ! to be the multi-process slot number. The slot number varies between 1 and the number of concurrent jobs permitted by the -j flag given on the command line. Armed with this, it is possible to write a multiple host shell. For example:

```
#!/bin/sh

# -This -sample -JAMSHELL -uses -the -SunOS -on(1) -command -to -execute -a
# -command -string -with -an -identical -environment -on -another -host.

# -Set -JAMSHELL -= -jamshell -! -%
#
# -where -jamshell -is -the -name -of -this -shell -file.
#
# -This -version -handles -up -to --j6; -after -that -they -get -executed
# -locally.

case -$1 -in
1|4) -on -winken -sh --c -"$2";;
2|5) -on -blinken -sh --c -"$2";;
3|6) -on -nod -sh --c -"$2";;
*) -eval -"$2";;
esac
```

__TIMING_RULE__ and __ACTION_RULE__

The __TIMING_RULE__ and __ACTION_RULE__ can be set to the name of a rule for bjam to call **after** an action completes for a target. They both give diagnostic information about the action that completed. For __TIMING_RULE__ the rule is called as:

```
rule -timing-rule -( -args -* -: -target -: -start -end -user -system -)
```

And __ACTION_RULE__ is called as:

```
 -out
```

```
? -)
```

The arguments for both are:

`args`	Any values following the rule name in the `__TIMING_RULE__` or `__ACTION_RULE__` are passed along here.
`target`	The `bjam` target that was built.
`command`	The text of the executed command in the action body.
`status`	The integer result of the executed command.
`start`	The starting timestamp of the executed command as a ISO-8601 UTC value.
`end`	The completion timestamp of the executed command as a ISO-8601 UTC value.
`user`	The number of user CPU seconds the executed command spent as a floating point value.
`system`	The number of system CPU seconds the executed command spent as a floating point value.
`output`	The output of the command as a single string. The content of the output reflects the use of the `-pX` option.

Note

If both variables are set for a target both are called, first `__TIMING_RULE__` then `__ACTION_RULE__`.

Modules

Boost Jam introduces support for modules, which provide some rudimentary namespace protection for rules and variables. A new keyword, "`module`" was also introduced. The features described in this section are primitives, meaning that they are meant to provide the operations needed to write Jam rules which provide a more elegant module interface.

Declaration

```
module -expression -{ -... -}
```

Code within the `{ ... }` executes within the module named by evaluating expression. Rule definitions can be found in the module's own namespace, and in the namespace of the global module as *module-name.rule-name*, so within a module, other rules in that module may always be invoked without qualification:

```
module -my_module
{
 - - - -rule -salute -( -x -) -{ -ECHO -$(x), -world -; -}
 - - - -rule -greet -( -) -{ -salute -hello -; -}
 - - - -greet -;
}
my_module.salute -goodbye -;
```

When an invoked rule is not found in the current module's namespace, it is looked up in the namespace of the global module, so qualified calls work across modules:

```
module -your_module
{
 - - - -rule -bedtime -( -) -{ -my_module.salute -goodnight -; -}
```

```
}
```

Variable Scope

Each module has its own set of dynamically nested variable scopes. When execution passes from module A to module B, all the variable bindings from A become unavailable, and are replaced by the bindings that belong to B. This applies equally to local and global variables:

```
module -A
{
 - - - -x -= -1 -;
 - - - -rule -f -( -)
 - - - -{
 - - - - - - - -local -y -= -999 -; -# -becomes -visible -again -when -B.f -calls -A.g
 - - - - - - - -B.f -;
 - - - -}
 - - - -rule -g -( -)
 - - - -{
 - - - - - - - -ECHO -$(y) -; - - - - -# -prints -"999"
 - - - -}
}
module -B
{
 - - - -y -= -2 -;
 - - - -rule -f -( -)
 - - - -{
 - - - - - - - -ECHO -$(y) -; -# -always -prints -"2"
 - - - - - - - -A.g -;
 - - - -}
}
```

The only way to access another module's variables is by entering that module:

```
rule -peek -( -module-name -? -: -variables -+ -)
{
 - - - -module -$(module-name)
 - - - -{
 - - - - - - - -return -$($(>)) -;
 - - - -}
}
```

Note that because existing variable bindings change whenever a new module scope is entered, argument bindings become unavailable. That explains the use of "`$(>)`" in the peek rule above.

Local Rules

```
local -rule -rulename...
```

The rule is declared locally to the current module. It is not entered in the global module with qualification, and its name will not appear in the result of:

```
[ -RULENAMES -module-name -]
```

The RULENAMES Rule

```
rule -RULENAMES -( -module -? -)
```

Returns a list of the names of all non-local rules in the given module. If *module* is omitted, the names of all non-local rules in the global module are returned.

The VARNAMES Rule

```
rule -VARNAMES -( -module -? -)
```

Returns a list of the names of all variable bindings in the given module. If *module* is omitted, the names of all variable bindings in the global module are returned.

Note

This includes any local variables in rules from the call stack which have not returned at the time of the VARNAMES invocation.

The IMPORT Rule

IMPORT allows rule name aliasing across modules:

```
rule -IMPORT -( -source_module -? -: -source_rules -*
 - - - - - - - - - - - -: -target_module -? -: -target_rules -* -)
```

The IMPORT rule copies rules from the *source_module* into the *target_module* as local rules. If either *source_module* or *target_module* is not supplied, it refers to the global module. *source_rules* specifies which rules from the *source_module* to import; *target_rules* specifies the names to give those rules in *target_module*. If *source_rules* contains a name which doesn't correspond to a rule in *source_module*, or if it contains a different number of items than *target_rules*, an error is issued. For example,

```
# -import -m1.rule1 -into -m2 -as -local -rule -m1-rule1.
IMPORT -m1 -: -rule1 -: -m2 -: -m1-rule1 -;
# -import -all -non-local -rules -from -m1 -into -m2
IMPORT -m1 -: -[ -RULENAMES -m1 -] -: -m2 -: -[ -RULENAMES -m1 -] -;
```

The EXPORT Rule

EXPORT allows rule name aliasing across modules:

```
rule -EXPORT -( -module -? -: -rules -* -)
```

The EXPORT rule marks *rules* from the source_module as non-local (and thus exportable). If an element of *rules* does not name a rule in *module*, an error is issued. For example,

```
module -X -{
 - -local -rule -r -{ -ECHO -X.r -; -}
}
IMPORT -X -: -r -: -: -r -; -# -error -- -r -is -local -in -X
EXPORT -X -: -r -;
IMPORT -X -: -r -: -: -r -; -# -OK.
```

The CALLER_MODULE Rule

```
rule -CALLER_MODULE -( -levels -? -)
```

CALLER_MODULE returns the name of the module scope enclosing the call to its caller (if levels is supplied, it is interpreted as an integer number of additional levels of call stack to traverse to locate the module). If the scope belongs to the global module, or if no such module exists, returns the empty list. For example, the following prints "{Y} {X}":

```
module -X -{
 - - - -rule -get-caller -{ -return -[ -CALLER_MODULE -] -; -}
 - - - -rule -get-caller's-caller -{ -return -[ -CALLER_MODULE -1 -] -; -}
 - - - -rule -call-Y -{ -return -Y.call-X2 -; -}
}
module -Y -{
 - - - -rule -call-X -{ -return -X.get-caller -; -}
 - - - -rule -call-X2 -{ -return -X.get-caller's-caller -; -}
}
callers -= -[ -X.get-caller -] -[ -Y.call-X -] -[ -X.call-Y -] -;
ECHO -{$(callers)} -;
```

The DELETE_MODULE Rule

```
rule -DELETE_MODULE -( -module -? -)
```

DELETE_MODULE removes all of the variable bindings and otherwise-unreferenced rules from the given module (or the global module, if no module is supplied), and returns their memory to the system.

Note

Though it won't affect rules that are currently executing until they complete, DELETE_MODULE should be used with extreme care because it will wipe out any others and all variable (including locals in that module) immediately. Because of the way dynamic binding works, variables which are shadowed by locals will not be destroyed, so the results can be really unpredictable.

Miscellaneous

Diagnostics

In addition to generic error messages, bjam may emit one of the following:

```
warning: -unknown -rule -X
```

A rule was invoked that has not been defined with an "actions" or "rule" statement.

```
using -N -temp -target(s)
```

Targets marked as being temporary (but nonetheless present) have been found.

```
updating -N -target(s)
```

Targets are out-of-date and will be updated.

```
can't -find -N -target(s)
```

Source files can't be found and there are no actions to create them.

```
can't -make -N -target(s)
```

Due to sources not being found, other targets cannot be made.

```
warning: -X -depends -on -itself
```

A target depends on itself either directly or through its sources.

```
don't -know -how -to -make -X
```

A target is not present and no actions have been defined to create it.

```
X -skipped -for -lack -of -Y
```

A source failed to build, and thus a target cannot be built.

```
warning: -using -independent -target -X
```

A target that is not a dependency of any other target is being referenced with `$(<)` or `$(>)`.

```
X -removed
```

`BJam` removed a partially built target after being interrupted.

Bugs, Limitations

For parallel building to be successful, the dependencies among files must be properly spelled out, as targets tend to get built in a quickest-first ordering. Also, beware of un-parallelizable commands that drop fixed-named files into the current directory, like `yacc(1)` does.

A poorly set `$(JAMSHELL)` is likely to result in silent failure.

Fundamentals

This section is derived from the official Jam documentation and from experience using it and reading the Jambase rules. We repeat the information here mostly because it is essential to understanding and using Jam, but is not consolidated in a single place. Some of it is missing from the official documentation altogether. We hope it will be useful to anyone wishing to become familiar with Jam and the Boost build system.

- Jam "`rules`" are actually simple procedural entities. Think of them as functions. Arguments are separated by colons.
- A Jam **target** is an abstract entity identified by an arbitrary string. The build-in `DEPENDS` rule creates a link in the dependency graph between the named targets.
- Note that the original Jam documentation for the built-in `INCLUDES` rule is incorrect: `INCLUDES` *`targets1`* `:` *`targets2`* causes everything that depends on a member of *targets1* to depend on all members of *targets2*. It does this in an odd way, by tacking *targets2* onto a special tail section in the dependency list of everything in *targets1*. It seems to be OK to create circular dependencies this way; in fact, it appears to be the "right thing to do" when a single build action produces both *targets1* and *targets2*.

- When a rule is invoked, if there are `actions` declared with the same name as the rule, the actions are added to the updating actions for the target identified by the rule's first argument. It is actually possible to invoke an undeclared rule if corresponding actions are declared: the rule is treated as empty.

- Targets (other than `NOTFILE` targets) are associated with paths in the file system through a process called binding. Binding is a process of searching for a file with the same name as the target (sans grist), based on the settings of the target-specific `SEARCH` and `LOCATE` variables.

- In addition to local and global variables, jam allows you to set a variable `on` a target. Target-specific variable values can usually not be read, and take effect only in the following contexts:

  - In updating actions, variable values are first looked up `on` the target named by the first argument (the target being updated). Because Jam builds its entire dependency tree before executing actions, Jam rules make target-specific variable settings as a way of supplying parameters to the corresponding actions.

  - Binding is controlled *entirely* by the target-specific setting of the `SEARCH` and `LOCATE` variables, as described here.

  - In the special rule used for header file scanning, variable values are first looked up `on` the target named by the rule's first argument (the source file being scanned).

- The "bound value" of a variable is the path associated with the target named by the variable. In build actions, the first two arguments are automatically replaced with their bound values. Target-specific variables can be selectively replaced by their bound values using the `bind` action modifier.

- Note that the term "binding" as used in the Jam documentation indicates a phase of processing that includes three sub-phases: *binding* (yes!), update determination, and header file scanning. The repetition of the term "binding" can lead to some confusion. In particular, the Modifying Binding section in the Jam documentation should probably be titled "Modifying Update Determination".

- "Grist" is just a string prefix of the form <*characters*>. It is used in Jam to create unique target names based on simpler names. For example, the file name "`test.exe`" may be used by targets in separate subprojects, or for the debug and release variants of the "same" abstract target. Each distinct target bound to a file called "test.exe" has its own unique grist prefix. The Boost build system also takes full advantage of Jam's ability to divide strings on grist boundaries, sometimes concatenating multiple gristed elements at the beginning of a string. Grist is used instead of identifying targets with absolute paths for two reasons:

  1. The location of targets cannot always be derived solely from what the user puts in a Jamfile, but sometimes depends also on the binding process. Some mechanism to distinctly identify targets with the same name is still needed.

  2. Grist allows us to use a uniform abstract identifier for each built target, regardless of target file location (as allowed by setting ALL_LOCATE_TARGET).

- When grist is extracted from a name with $(var:G), the result includes the leading and trailing angle brackets. When grist is added to a name with $(var:G=expr), existing grist is first stripped. Then, if expr is non-empty, leading <s and trailing >s are added if necessary to form an expression of the form <expr2>; <expr2> is then prepended.

- When Jam is invoked it imports all environment variable settings into corresponding Jam variables, followed by all command-line (-s...) variable settings. Variables whose name ends in PATH, Path, or path are split into string lists on OS-specific path-list separator boundaries (e.g. ":" for UNIX and ";" for Windows). All other variables are split on space (" ") boundaries. Boost Jam modifies that behavior by allowing variables to be quoted.

- A variable whose value is an empty list or which consists entirely of empty strings has a negative logical value. Thus, for example, code like the following allows a sensible non-empty default which can easily be overridden by the user:

```
MESSAGE -?\= -starting -jam... -;
if -$(MESSAGE) -{ -ECHO -The -message -is: -$(MESSAGE) -; -}
```

  If the user wants a specific message, he invokes jam with `"-sMESSAGE=message text"`. If he wants no message, he invokes jam with `-sMESSAGE=` and nothing at all is printed.

- The parsing of command line options in Jam can be rather unintuitive, with regards to how other Unix programs accept options. There are two variants accepted as valid for an option:

  1. `-xvalue`, and

2. `-x value`.

History

3.1.17

A year in the making this release has many stability improvements and various performance improvements. And because of the efforts of Jurko the code is considerably more readable!

- Reflect the results of calling bjam from Python. -- *Rene R.*
- For building on Windows: Rework how arguments are parsed and tested to fix handling of quoted arguments, options arguments, and arguments with "=". -- *Rene R.*
- Try to work around at least one compiler bug with GCC and variable aliasing that causes crashes with hashing file cache entries. -- *Rene R.*
- Add -Wc,-fno-strict-aliasing for QCC/QNX to avoid the same aliasing crashes as in the general GCC 4.x series (thanks to Niklas Angare for the fix). -- *Rene R.*
- On Windows let the child bjam commands inherit stdin, as some commands assume it's available. -- *Rene R.*
- On Windows don't limit bjam output to ASCII as some tools output characters in extended character sets. -- *Rene R.*
- Isolate running of bjam tests to individual bjam instances to prevent possible spillover errors from one test affecting another test. Separate the bjam used to run the tests vs. the bjam being tested. And add automatic re-building of the bjam being tested. -- *Rene R.*
- Fix some possible overrun issues revealed by Fortify build. Thanks to Steven Robbins for pointing out the issues. -- *Rene R.*
- Handle \n and \r escape sequences. -- *Vladimir P.*
- Minor edits to remove -Wall warnings. -- *Rene R.*
- Dynamically adjust pwd buffer query size to allow for when PATH_MAX is default defined instead of being provided by the system C library. -- *Rene R.*
- Minor perf improvement for bjam by replacing hash function with faster version. Only 1% diff for Boost tree. -- *Rene R.*
- Updated Boost Jam's error location reporting when parsing Jamfiles. Now it reports the correct error location information when encountering an unexpected EOF. It now also reports where an invalid lexical token being read started instead of finished which makes it much easier to find errors like unclosed quotes or curly braces. -- *Jurko G.*
- Removed the -xarch=generic architecture from build.jam as this option is unknown so the Sun compilers on Linux. -- *Noel B.*
- Fixed a bug with T_FATE_ISTMP getting reported as T_FATE_ISTMP & T_FATE_NEEDTMP at the same time due to a missing break in a switch statement. -- *Jurko G.*
- Fixed a Boost Jam bug causing it to sometimes trigger actions depending on targets that have not been built yet. -- *Jurko G.*
- Added missing documentation for Boost Jam's :T variable expansion modifier which converts all back-slashes ('\') to forward slashed ('/'). -- *Jurko G.*

- Added Boost Jam support for executing command lines longer than 2047 characters (up to 8191) characters when running on Windows XP or later OS version. -- *Jurko G.*
- Fixed a Boost Jam bug on Windows causing its SHELL command not to work correctly with some commands containing quotes. -- *Jurko G.*
- Corrected a potential memory leak in Boost Jam's builtin_shell() function that would appear should Boost Jam ever start to release its allocated string objects. -- *Jurko G.*
- Made all Boost Jam's ECHO commands automatically flush the standard output to make that output more promptly displayed to the user. -- *Jurko G.*
- Made Boost Jam tests quote their bjam executable name when calling it allowing those executables to contain spaces in their name and/or path. -- *Jurko G.*
- Change execunix.c to always use fork() instead of vfork() on the Mac. This works around known issues with bjam on PPC under Tiger and a problem reported by Rene with bjam on x86 under Leopard. -- *Noel B.*
- Corrected a bug in Boost Jam's base Jambase script causing it to trim the error message displayed when its boost-build rule gets called multiple times. -- *Jurko G.*
- When importing from Python into an module with empty string as name, import into root module. -- *Vladimir P.*
- Patch for the NORMALIZE_PATH builtin Boost Jam rule as well as an appropriate update for the path.jam Boost Build module where that rule was being used to implement path join and related operations. -- *Jurko G.*
- Fixed a bug causing Boost Jam not to handle target file names specified as both short and long file names correctly. -- *Jurko G.*
- Relaxed test, ignoring case of drive letter. -- *Roland S.*
- Implemented a patch contributed by Igor Nazarenko reimplementing the list_sort() function to use a C qsort() function instead of a hand-crafted merge-sort algorithm. Makes some list sortings (e.g. 1,2,1,2,1,2,1,2, ...) extremely faster, in turn significantly speeding up some project builds. -- *Jurko G.*
- Fixed a bug with bjam not handling the " root Windows path correctly without its drive letter being specified. -- *Jurko G.*
- Solved the problem with child process returning the value 259 (Windows constant STILL_ACTIVE) causing bjam never to detect that it exited and therefore keep running in an endless loop. -- *Jurko G.*
- Solved the problem with bjam going into an active wait state, hogging up processor resources, when waiting for one of its child processes to terminate while not all of its available child process slots are being used. -- *Jurko G.*
- Solved a race condition between bjam's output reading/child process termination detection and the child process's output generation/termination which could have caused bjam not to collect the terminated process's final output. -- *Jurko G.*
- Change from vfork to fork for executing actions on Darwin to improve stability. -- *Noel B.*
- Code reformatting and cleanups. -- *Jurko G.*
- Implement ISFILE built-in. -- *Vladimir P.*

3.1.16

This is mostly a bug fix release.

- Work around some Windows CMD.EXE programs that will fail executing a totally empty batch file. -- *Rene R.*
- Add support for detection and building with `vc9`. -- *John P.*

- Plug memory leak when closing out actions. Thanks to Martin Kortmann for finding this. -- *Rene R.*
- Various improvements to `__TIMING_RULE__` and `__ACTION_RULE__` target variable hooks. -- *Rene R.*
- Change `JAMDATE` to use common ISO date format. -- *Rene R.*
- Add test for result status values of simple actions, i.e. empty actions. -- *Rene R.*
- Fix buffer overrun bug in expanding `@()` subexpressions. -- *Rene R.*
- Check empty string invariants, instead of assuming all strings are allocated. And reset strings when they are freed. -- *Rene R.*
- Add `OSPLAT=PARISC` for HP-UX PA-RISC. -- *Boris G.*
- Make quietly actions really quiet by not printing the command output. The output for the quietly actions is still available through `__ACTION_RULE__`. -- *Rene R.*
- Switch intel-win32 to use static multi thread runtime since the single thread static runtime is no longer available. -- *Rene R.*
- When setting `OSPLAT`, check `__ia64` macro. -- *Boris G.*
- Get the unix timing working correctly. -- *Noel B.*
- Add `-fno-strict-aliasing` to compilation with gcc. Which works around GCC-4.2 crash problems. -- *Boris G.*
- Increased support for Python integration. -- *Vladimir P.*, *Daniel W.*
- Allow specifying options with quotes, i.e. `--with-python=xyz`, to work around the CMD shell using = as an argument separator. -- *Rene R.*
- Add values of variables specified with -s to .EVNRION module, so that we can override environment on command line. -- *Vladimir P.*
- Make NORMALIZE_PATH convert \ to /. -- *Vladimir P.*

3.1.15

This release sees a variety of fixes for long standing Perforce/Jam problems. Most of them relating to running actions in parallel with the -jN option. The end result of the changes is that running parallel actions is now reliably possible in Unix and Windows environments. Many thanks to Noel for joining the effort, to implement and fix the Unix side of stuff.

- Add support for building bjam with pgi and pathscale toolsets. -- *Noel B.*
- Implement running action commands through pipes (-p option) to fix jumbled output when using parallel execution with -j option. This is implemented for Unix variants, and Windows (Win32/NT). -- *Rene R.*, *Noel B.*
- Add "sun" as alias to Sun Workshop compiler tools. -- *Rene R.*
- Set MAXLINE in jam.h to 23k bytes for AIX. The piecemeal archive action was broken with the default MAXLINE of 102400. Because the AIX shell uses some of the 24k default buffer size for its own use, I reduced it to 23k. -- *Noel B.*
- Make use of output dir options of msvc to not polute src dir with compiled files. -- *Rene R.*
- A small fix, so -d+2 will always show the "real" commands being executed instead of casually the name of a temporary batch file. -- *Roland S.*
- Add test to check 'bjam -n'. -- *Rene R.*

- Add test to check 'bjam -d2'. -- *Rene R.*
- Bring back missing output of -n option. The -o option continues to be broken as it has been for a long time now because of the @ file feature. -- *Rene R.*
- Update GC support to work with Boehm GC 7.0. -- *Rene R.*
- Revert the BOOST_BUILD_PATH change, since the directory passed to boost-build should be first in searched paths, else project local build system will not be picked correctly. The order had been changed to allow searching of alternate user-config.jam files from boost build. This better should be done with --user-config= switch or similar. -- *Roland S.*
- Initial support for defining action body from Python. -- *Vladimir P.*
- Implement @() expansion during parse phase. -- *Rene R.*
- Define OSPLAT var unconditionally, and more generically, when possible. -- *Rene R.*
- Fix undeclared INT_MAX on some platforms, i.e. Linux. -- *Rene R.*
- Modified execunix.c to add support for terminating processes that consume too much cpu or that hang and fail to consume cpu at all. This in support of the bjam -lx option. -- *Noel B.*
- Add internal dependencies for multi-file generating actions to indicate that the targets all only appear when the first target appears. This fixes the long standing problem Perforce/Jam has with multi-file actions and parallel execution (-jN). -- *Rene R.*
- Add test of -l limit option now that it's implemented on windows and unix. -- *Rene R.*
- Add test for no-op @() expansion. -- *Rene R.*
- Handle invalid formats of @() as doing a straight substitution instead of erroring out. -- *Rene R.*
- Various fixes to compile on SGI/Irix. -- *Noel B.*
- Add output for when actions timeout with -lN option. -- *Rene R., Noel B.*
- Add needed include (according to XOPEN) for definition of WIFEXITED and WEXITSTATUS. -- *Markus S.*

Appendix B. Differences to Boost.Build V1

While Boost.Build V2 is based on the same ideas as Boost.Build V1, some of the syntax was changed, and some new important features were added. This chapter describes most of the changes.

Configuration

In V1, toolsets were configured by environment variables. If you wanted to use two versions of the same toolset, you had to create a new toolset module that would set the variables and then invoke the base toolset. In V2, toolsets are configured by the `using`, and you can easily configure several versions of a toolset. See the section called “Configuration” for details.

Writing Jamfiles

Probably one of the most important differences in V2 Jamfiles is the use of project requirements. In V1, if several targets had the same requirements (for example, a common `#include` path), it was necessary to manually write the requirements or use a helper rule or template target. In V2, the common properties can be specified with the `requirements` project attribute, as documented in the section called “Projects”.

Usage requirements also help to simplify Jamfiles. If a library requires all clients to use specific `#include` paths or macros when compiling code that depends on the library, that information can be cleanly represented.

The difference between `lib` and `dll` targets in V1 is completely eliminated in V2. There's only one library target type, `lib`, which can create either static or shared libraries depending on the value of the `<link>` feature. If your target should be only built in one way, you can add `<link>shared` or `<link>static` to its requirements.

The syntax for referring to other targets was changed a bit. While in V1 one would use:

```
exe -a -: -a.cpp -<lib>../foo/bar -;
```

the V2 syntax is:

```
exe -a -: -a.cpp -../foo//bar -;
```

Note that you don't need to specify the type of other target, but the last element should be separated from the others by a double slash to indicate that you're referring to target `bar` in project `../foo`, and not to project `../foo/bar`.

Build process

The command line syntax in V2 is completely different. For example

```
bjam --sTOOLS=msvc --sBUILD=release -some_target
```

now becomes:

```
bjam -toolset=msvc -variant=release -some_target
```

or, using implicit features, just:

```
bjam -msvc -release -some_target
```

See the reference for a complete description of the syntax.

Index

Symbols

64-bit compilation,
  gcc, 39
  Microsoft Visual Studio, 41
  Sun Studio, 46

C

common signature, 19
cross compilation, 33

E

embed-manifest, 38
exe, 26

F

fat binaries, 39
features
  builtin, 35

G

generators, 53
glob-tree,

I

install-source-root, 29
instruction-set,

M

main target (see metataget)
  declaration syntax, 19
manifest file
  embedding, 38
metatarget
  definition, 12

P

preserve-test-targets, 31
property
  definition, 12
  propagation, 12

R

requirements, 20
rule , 12
runtime linking,

S

STLport, 46